COURSE NOTES

Programming 2

COSC1076/2207

MAJOR CONTRIBUTORS: DR CASPAR RYAN, CHARLES THEVATHAYAN, PETER TILMANIS

Course Leader/Lecturer: Dr Caspar Ryan

SCHOOL OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
RMIT UNIVERSITY, MELBOURNE, AUSTRALIA.

Semester 1, 2010

RMIT UNIVERSITY Computer Science and Information Technology

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 1: Object-Oriented Development

Lecture Slides

Object-Oriented Development

- In general, ***development*** consists of three broad activities
  - These can be further subdivided but conceptually activities generally fall into one of these broader categories
- **Analysis (e.g. OOA)**
  - understanding and defining the problem
- **Design (e.g. OOD)**
  - A high level conceptual solution expressed using diagrams, pseudocode etc.
- **Programming (e.g. OOP)**
  - A detailed solution implemented in a programming language such as Java, C# or C++

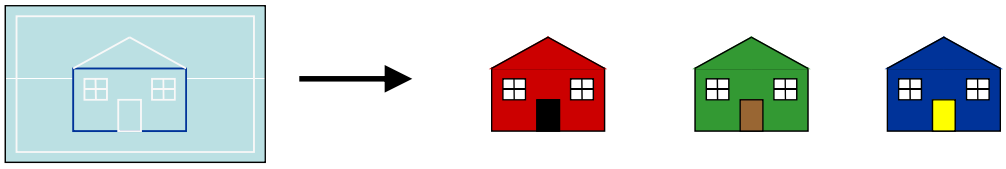
Background

- In *Programming 1* you have done mainly OOP
- In *Programming 2* you will do OOP and some introductory OOD
- OOA is covered in software engineering
- OOD is covered in more detail in software engineering, advanced electives and Programming 3

Revision: Classes and Objects

A class can be compared to a **blueprint** created by an architect when designing a house. It defines the important characteristics of the house, such as its walls, windows, electrical outlets, and so on.

Once we have the blueprint, several unique houses can be built, each with different addresses, furniture, colours, etc.



A 'blueprint' for a Java class contains attributes (variables) and methods. When an object is made out of the class, an **instance** of the class is said to be made (i.e. The class is *instantiated*). Unless explicitly specified (via static keyword), all of the properties of each instance are unique; for example, changing one instance's variable will not affect the other instances.

Revision: Object-Orientation in Java

Java is a purely object-oriented language: (almost) everything is an object.

- Java application programs are **built only of classes**
- Java provides a **very small core language** of primitive data types, operators and control structures; the remainder of the functionality is provided through a **very large class library** (the API.)
- Every primitive type in Java can be converted to an object, through the use of **wrapper classes**.

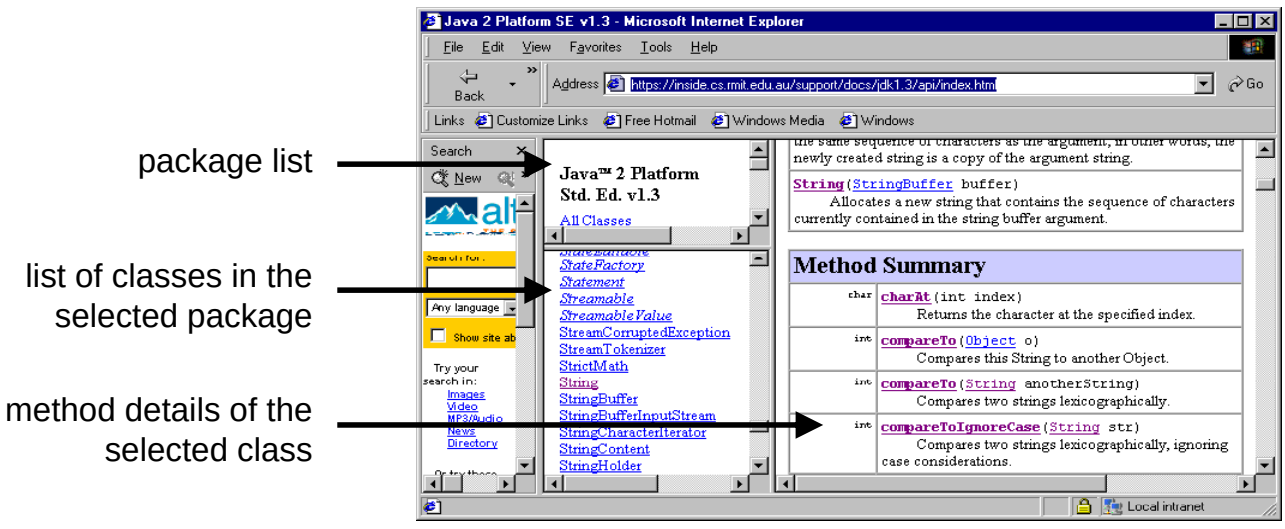
byte → Byte	float → Float	double → Double
Short → Short	long → Long	char → Character
int → Integer		boolean → Boolean

- Java supports all four key features of object programming; **encapsulation**, **inheritance**, **polymorphism** and **dynamic binding**.

Revision: Using the Java API Documentation

Java includes a large number of support classes with pre-defined functionality in its API (Application Programming Interface.)

The details of all these classes and their properties can be viewed easily using the Java Docs.



Example: a Bank System

Take the example of a **bank**.

- The **bank** stores data about its **customers**.
- A **customer** has a name, a customer number, a password and one or more accounts.
- An **account** is identified by an account number and has a balance.
- A customer can perform withdrawals, deposits and balance queries on their accounts.

How can we **discover classes** (and their relationships)?

How can we **discover attributes** (variables) **and methods for each class**?

Discovering Classes and Relationships (1)

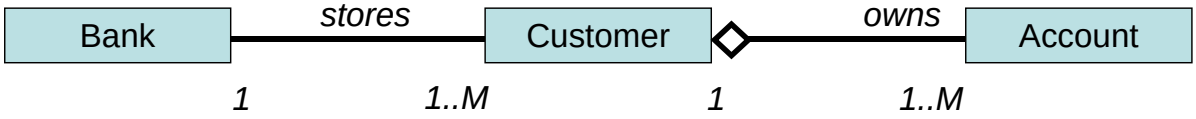
Classes are often either a **concrete entity** (a student, or a course), or an abstract concept (a shape, or form of transport.)

A good rule of thumb for discovering classes is to **look for nouns** in the problem specification.

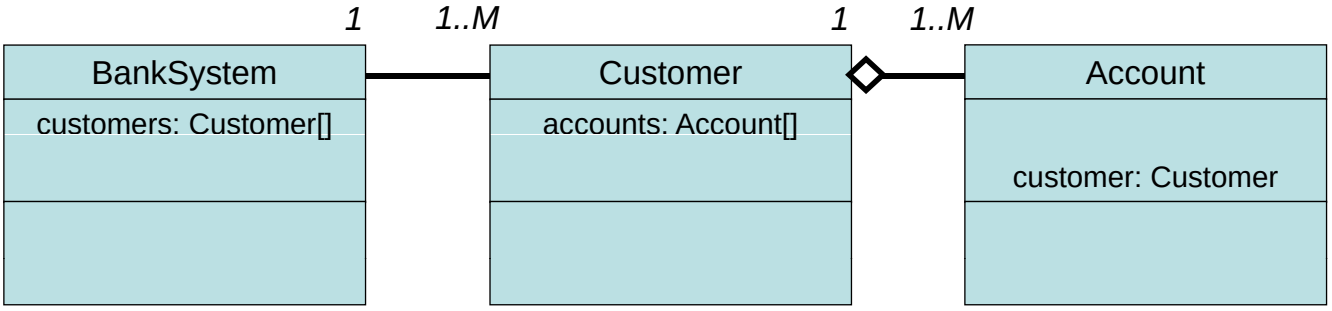
Class relationships can take different forms; **association**, **composition/aggregation** and **inheritance**.

In practice the difference between the first three and last form (inheritance) is the most important.

When determining cardinality (number of instances in a relationship e.g one to many or many to many) over-estimate rather than underestimate.



Representing Classes and Relationships (2)



Classes are represented by creating a new class type with the `class` keyword

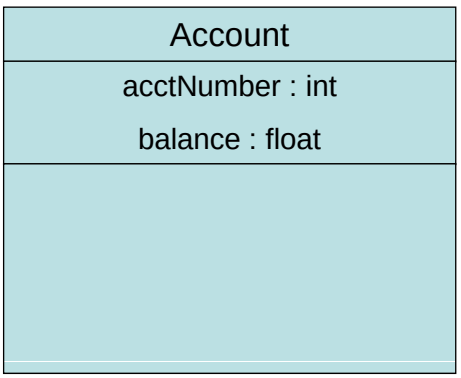
1..1 relationships are represented by a single attribute.

1..M relationships the M reference is held in the 1 class by some sort of collection (e.g. an array or vector). M .. M is basically a pair of 1 .. M collections.

Inheritance is represented using class inheritance in Java (`extends` keyword) and is potentially complex when using polymorphism, interfaces and abstract classes. However it is a powerful and sophisticated technique when done well!

This is covered in topics 2 and 3.

Discovering Attributes



To discover attributes, ask question such as:

"I am an account. What should I know? What do I need to remember?"

- my account number?
- my account balance?
- my owner?
- my bank?
- etc.

For each attribute, you need to decide on its name, data type, visibility (public, protected, private or default (package private)], also any modifiers i.e. should it be a constant (final) or variable, and having instance or class scope (static).

Discovering Methods and Relationships (1)

Account
acctNumber : int balance : float
withdraw(amount : float) : boolean deposit(amount : float) getAcctNumber() : integer getBalance() : float

Tip: look for verbs in the problem specification.

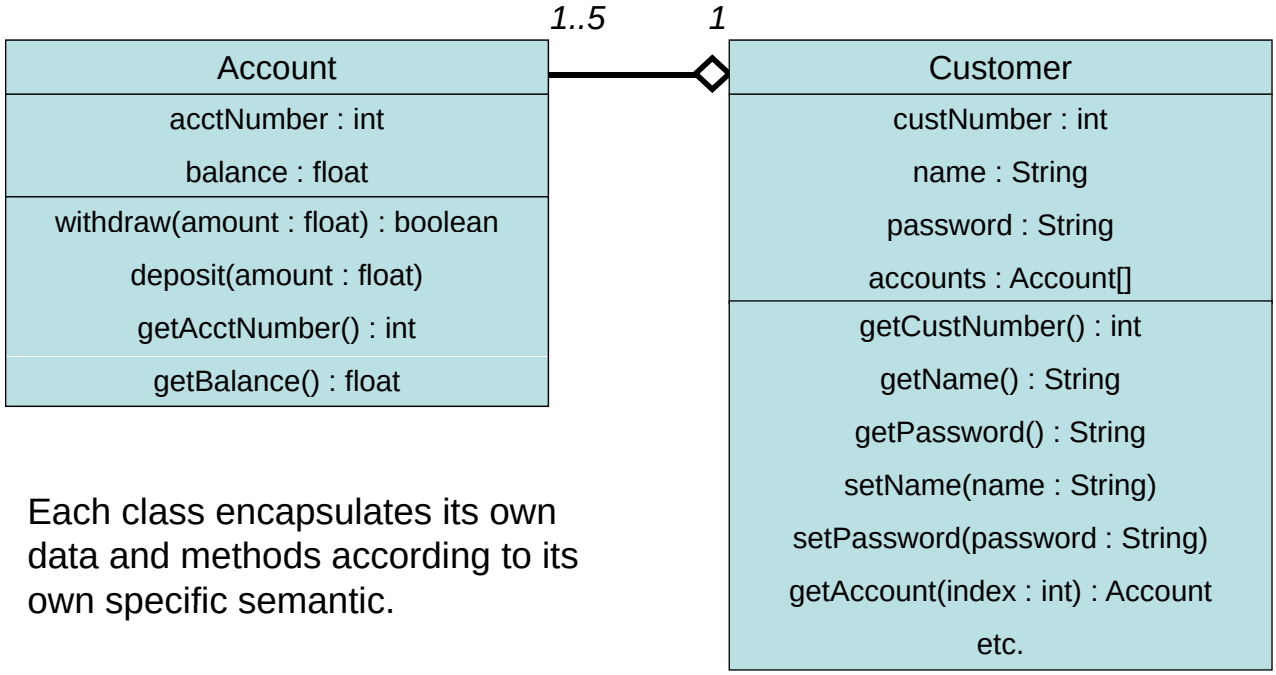
Ask question such as:

"I am an account. What services should I provide?" "What expertise should I have?" "What is my role?" "Who will be using me?" "Who do I interact with?"

- query my account number?
- query my account balance?
- change my account number?
- change my balance?
- deposit or withdraw money?
- etc.

Like attributes, for each method you also need to decide on its name, data types (for parameters and return value), visibility (public, protected, private or default (package)], and whether it should be final (not overridable) or static (called at the class level).

Discovering Methods and Relationships (2)



Each class encapsulates its own data and methods according to its own specific semantic.

The Account class: `Account.java` (1)

```
class Account {
  private int acctNumber = 0;
  private float balance = 0.0;

  public Account(int aNumber, float aBalance) {
    acctNumber = aNumber;
    balance = aBalance;
  }

  public int getAcctNumber() {
    return acctNumber;
  }
```

Variable declaration and initialisation

Class constructor. It has **no** return type, not even **`void`**.

Method definition

(in this case, an accessor method)

Note the use of **`private`** and **`public`**; encapsulation requires the support of **information hiding**, where internal implementation details must be kept hidden from the outside world. This is done with **visibility modifiers**.

The Account class: `Account.java` (2)

```
  public boolean withdraw(float amount){
    if (balance >= amount) {
      balance -= amount;
      return true;
    }
    else
      return false;
  }

  public void deposit(float amount) {
    balance += amount;
  }

  // other Account class methods would be added here
}
```

`withdraw()` returns a boolean to return the success of the operation. This is just a design decision. What else could be done?

The Constructor

The constructor is a **special method** that is invoked to create a new object. Its purpose is to **initialise the object** (including its instance variables) **to a known state**.

The constructor has the same name as the class. It is always invoked with the **new** keyword. It **never** has a return type.

```
Account a = new Account();
```

By default, every class is provided with a default constructor that takes no parameter. If a class provides its own constructor (that takes parameters), then the default constructor is **not** automatically provided.

In this case, the class ***must explicitly define*** the default constructor, if desired. Many classes have more than one constructor, each with a different parameter list (i.e. the constructors are overloaded.)

Accessors and Mutators

Data should be kept private to classes (in order to comply with information hiding.) In order to provide access to private data, we may need to provide accessor and mutator methods (also known as 'get' and 'set' methods, respectively.)

Accessors ('get' methods) retrieve the value of the attribute being accessed.

Mutators ('set' methods) modify the value of the attribute being accessed.

Accessors and mutators are usually provided for attributes, however in some cases it makes better sense to omit either (or both.) For example, should we provide a mutator for an account number?

Rule of thumb: provide accessors and mutators only when required, and when they make sense for a problem at hand.

Example Customer class: `Customer.java` (1)

```
class Customer {
  private String name = null;
  private int custNo = 0;
  private String password = null;

  private int numAccounts = 0;

  private Account[] accounts = new Account[MAX_ACCTS];

  private static final int MAX_ACCTS = 5;

  Customer(String name, int number){
    this.name = name;
    this.number = number;
  }

  public String getName(){
    return name;
  }
```

Keep track of how many accounts have been open so far

An array of `Account` objects

`final` makes it constant.

`static` makes it a class, rather than instance, attribute.

The Customer class: `Customer.java` (2)

```
  public int getCustNo(){
    return custNo;
  }

  public String getPassword(){
    return password;
  }

  public Account getAccount(int index){
    return accounts[index];
  }

  public Account[] getAccounts(){
    return accounts;
  }
```

Return only the account at a given index

Return the whole array of accounts

The Customer class: `Customer.java` (3)

```
    public boolean addAccount(Account a) {
      if (numAccounts < MAX_ACCOUNTS){
        accounts[numAccounts] = a;
        numAccounts++;
        return true;
      }
      else
        return false;
    }

    public void setPassword(String pw){
      password = pw;
    }

    // other methods go here
}
```

Account successfully added

Account array failed as it exceeded the array limit

Driver Program to Test the `Customer` Class

```
class Driver {
  public static void main(String[] args){
    Customer john = new Customer("John", 1234);
    Account jAcct = new Account(1, 300);
    john.addAccount(jAcct);

    Account returned = john.getAccount(1);
    System.out.println("John's account balance: " +
                            returned.getBalance());
  }
}
```

The driver program (main method) acts like a controller. All activities start and eventually end here.

Objects communicate by sending messages to each other.

Every method (and control structure within) should be tested * testing covered in more detail in software engineering course.

A Simple Bank System: `Bank.java`

```
class Bank {
  public static void main(String[] args){

    if (args.length < 2){
      System.err.println("Usage: java Bank user bal");
      System.exit(1);
    }

    Customer cust = new Customer(args[0], 1);
    float balance = Float.parseFloat(args[1]);

    cust.addAccount(new Account(1, balance));
    System.out.println("New account created!");

    Account ret = cust.getAccount(0);
    System.out.println("Balance: " + ret.getBalance());
  }
}
```

This holds the command line arguments. This program requires two.

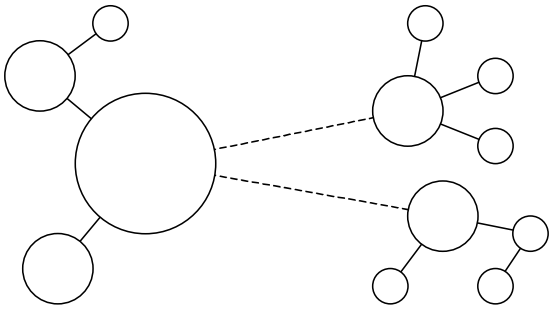
Parsing a `String` to a `float`.

Shorthand

Object-Oriented Design

As we have seen **object-oriented programming** is based on a **number of objects working together** to perform a function.

This kind of approach to developing code can give a number of benefits, such as easier **code maintenance** and enhanced **re-usability**.



a developers' object-oriented program

existing, re-usable software components

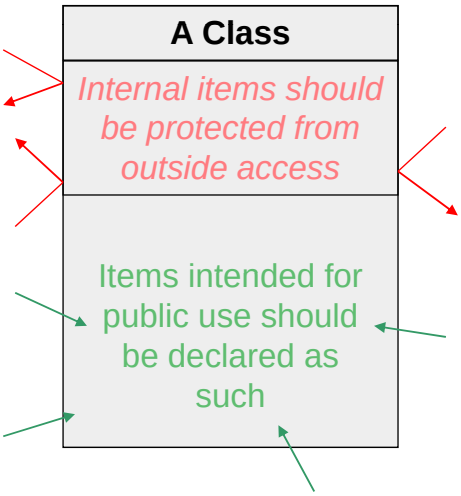
However, a **poorly designed** object-oriented program can be **just as, if not more difficult to use and maintain** than if it were developed using procedural or functional approaches.

Information Hiding

Now that all of Java's object capabilities have been detailed, we can review the fundamentals of **good OO design** with a **focus on implementation**.

Information hiding is a fundamental requirement to good OO design.

It means that outside access to the class' internals should be on a **"need to know basis"** – if it is not expected for a particular data item or feature to be publicly available, its visibility should be restricted.

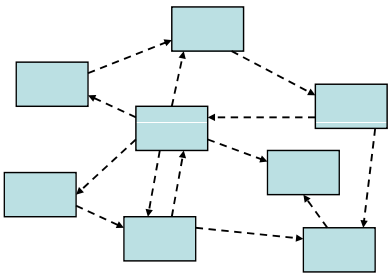


Coupling

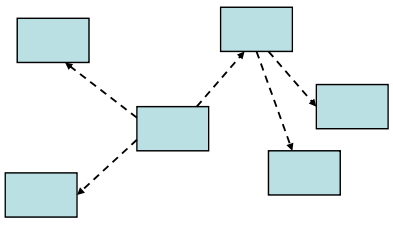
Classes should also be properly **encapsulated**. This refers to each class holding all the data/functionality it requires to fulfil its purpose, without being unduly dependent on others.

The dependence of some classes on others in this context is called **coupling**. This reliance should be minimised in your class designs wherever possible.

Excessive coupling between classes **can create problems** when items are modified – with so many dependencies, changing one will imply that many others may also need to be changed.



excessively high coupling



low coupling

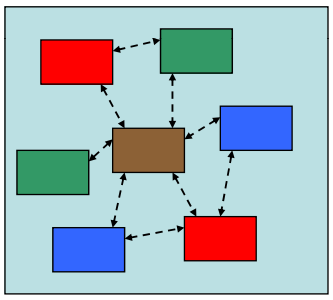
Cohesion

When a class is designed, each of its elements should have a **logical and well-defined role** in the class being able to fulfil its purpose. This **tight relation** of internal components is known as **cohesion**.

A **highly cohesive object** will **not** have items that are **only loosely related** to its role – the object will define **one** component well, and not attempt to be a “Jack of all trades”.

Cohesion is a **good** thing – it reflects a design that is **very clear and pure** to its intended purpose.

Coupling and Cohesion is a tradeoff – increasing cohesion (good) can increase coupling (bad).



A highly cohesive object will have elements that are closely inter-related.

Cohesion and Coupling Trade-off

- Cohesion and coupling are a trade-off
  - improving one may have a negative impact on the other
- e.g. Consider a program with only one class
  - it has low coupling (good) since there are no other classes to couple to
  - however cohesion will be low (bad) because the class will be responsible for everything
  - e.g. user interface, database management, unrelated domain logic (accounts, customers etc.)
  - => lots of unrelated methods = not cohesive
- Write cohesive classes first then try to minimise coupling

Cohesion and Coupling Trade-off (continued)

- Conversely imagine a system with a large number of very small and highly cohesion classes that only have one method
  - One method means only a single functionality so cohesion is high (good).
  - However this system will be highly coupled (bad) because all the small classes have to interact to get work done

'Good' Object Oriented Program Design

An object oriented program design should:

- have all internal information well protected from others
- have classes that have a clear and focused purpose (maximise cohesion)
- consist of classes that are highly independent (minimise coupling)

Adhering to these basic design principles should help design object-oriented programs that are **easily understood**, **easily maintainable**, and consist of **highly reusable** components.

A good program design can **save a lot of time** when it comes to implementation, by eliminating the need for "hacking and patching" code and class models to make them work.

OOD Notation – A Diagrammatic Approach

Class diagrams show the structure of a program at the class level, showing

methods and attributes of classes

relationship between classes,

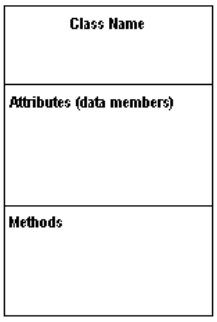
eg inheritance, aggregation

Can range from low (classes and relationships only) to high detail (e.g. full method signatures, visibility etc.)

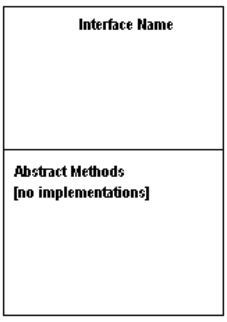
Class diagrams are part of a modelling language (UML) that you will encounter in later subjects in more detail.

OOD Notation – Class Diagram Approach (2)

A class representation in a class diagram

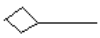


An interface representation in class diagram



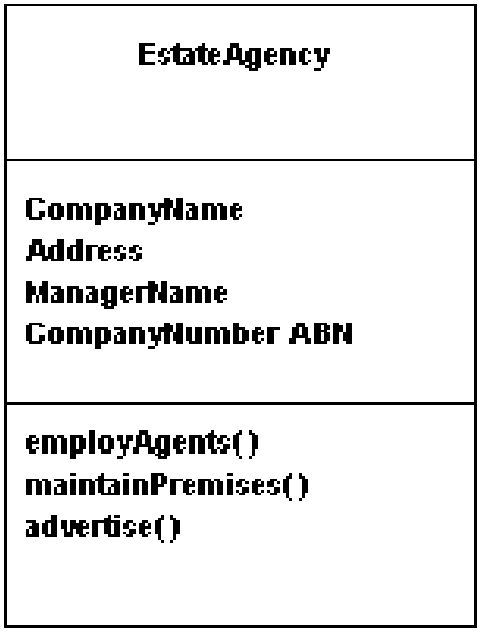
Inheritance
class-class
interface-interface

Association
a relationship
between classes



Aggregation / composition

OOD Example – Using diagrammatic approach



Exceptions and Error Handling

An exception defines an **unusual or erroneous situation** that can be handled by the programmer, for example an ArithmeticException (divide by zero), or an ArrayIndexOutOfBoundsException.

An error defines a generally **fatal machine-related problem** that cannot be handled by the programmer, so the program will terminate prematurely (crash.) For example, OutOfMemoryError.

Exception handling allows normal execution and exception execution flows to be separated. This **assists program design and implementation** as follows:

- **separation** allows normal execution to be made more efficient.
- some programs are required to **never** terminate abnormally.
- the **programmer can decide** where to handle an exception.
- it facilitates **debugging**.

Java Exceptions (1)

An exception is an object that can be returned from a method without the use of the return keyword. (An exception object is thrown, not returned.)

All objects that can be thrown are descendants of `java.lang.Throwable`. This class has two children, `java.lang.Error` and `java.lang.Exception`.

The class `java.lang.Exception` is very basic: the only data it holds is a single string, that can be used to hold an error message.

Classes that extend `Exception`, both library- and programmer-defined, usually maintain the same interface.

* `RuntimeExceptions` do not need to be caught or declared in method signature

Exceptions should be caught by the programmer – either:

- where the exception occurs, or
- elsewhere in the program (higher up in the method stack)
- If in doubt how to handle throw the exception rather than handle badly

Java Exceptions (2)

If an exception is not caught by the program, the virtual machine ultimately catches the exception, and terminates the program abnormally (it crashes.) When an exception is caught in this way, the error message, and the place where it occurred in the program, are output:

```
class Zero {
  public static void main(String[] args) {
      int a = 10;
      int b = 0;
      System.out.println( a/b );
   }
}
```

```
java.lang.ArithmeticException: / by zero
   at Zero.main(Zero.java:7)
```

The Exception Crash Message

```
java.lang.ArithmeticException: / by zero
   at Zero.main(Zero.java:7)
```

An exception crash message contains not only the message string (in this case, “/ by zero”), but also the method stack trace which specifies exactly where the error occurred, in the form method name (filename:line).

In general, a trace has more than one line.

Each line shows the method that was called to get to the method above, i.e. the trace represents the method calling hierarchy, and depicts the runtime stack.

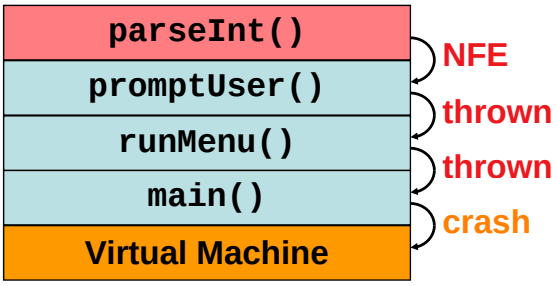
Exception Propagation

When an exception is thrown, it is moved along to the calling method in the method stack.

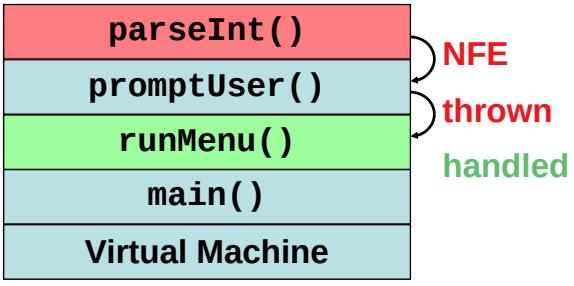
This process continues until it is either caught in a **`try..catch`** block, or it is thrown all the way to the Virtual Machine (which will cause the program to crash.)

The diagram on the right depicts how an exception can propagate through the method stack.

Without being caught:



Caught in `runMenu()`:



The `throws` Clause

The **`throws`** clause specifies that

(1) That exceptions are not going to be handled in the method where they are generated, and

(2) That they should be thrown to the calling method.

This has been used often in the past:

```
public static void main(String[] args) throws IOException {
   // code is here
}
```

All programs so far that included data input from the keyboard were defined as above.

The '**`throws IOException`**' tells Java that this method can throw an **`IOException`** to the calling method (that can be thrown from the **`readLine()`** method calls.)

The `try..catch` Block (1)

If, instead of crashing the program, we want to catch the exception and handle it, the exception-prone code can be isolated inside of a `try..catch` block, such as:

```
try {
  int a = Integer.parseInt("dog");
}
catch (NumberFormatException e) {
  System.err.println("Not an integer!");
}
```

If the call to **`parseInt()`** has a problem running, the **`NumberFormatException`** that it throws will make the **`catch`** block code run, instead of the exception being thrown.

The `try..catch` Block (2)

A **`try..catch`** block can have multiple **`catch`** statements.

However, due to the polymorphic behaviour of exception types, care must be taken to order them such that the most specific exception types are the first **`catch`** statements, and the more generic types caught last.

```
try {
   // source code here with
   // many possible exceptions
}
catch (NumberFormatException e){
   // handle this exception
}
catch (FileNotFoundException e){
   // handle this exception
}
catch (IOException e){
   // handle this exception
}
```


The `finally` Block

In some cases, there might be code that needs to be executed regardless of **whether or not** an operation completed successfully (for example, a file output stream that needs to be closed.)

A **`finally`** clause can be appended to the end of a **`try..catch`** block to allow this functionality.

```
try {
   // code
}
catch (exception-type) {
   // error handling
}
finally {
   // this code will always run
}
```

Creating an Exception Type (1)

To create a custom exception type, we can simply **extend** upon the appropriate subclass of the **`Exception`** hierarchy, and **inherit the properties** that make it usable as an exception.

For example, if our custom exception was to model a problem related to the process of inputting data, it would make logical sense to extend the **`IOException`** class.

All exceptions and errors inherit **basic error properties**, such as a **message** string and the **stack trace** information.

If those properties are all that we need (which is often the case), we simply extend the appropriate `Exception` class, and fill in the constructors.

Creating an Exception Type (2)

```
class RangeException extends Exception {

  RangeException() {
    super();
  }

  RangeException(String message) {
    super(message);
  }
}
```

Creating an Exception Type (2)

```
class RangeException extends Exception {

  RangeException() {
    super();
  }

  RangeException(String message) {
    super(message);
  }
}
```


Exceptions versus boolean return value

A boolean can be accidentally ignored, a checked exception must be caught and handled (and an unchecked exception still thrown at runtime).

A well named exception provides additional semantics/meaning about the exception whereas the boolean requires commenting/documentation to describe its purpose.

The boolean cannot distinguish between different error types, whereas you can use polymorphic catch blocks to catch multiple custom exceptions.

The main caution with exceptions is not to use them for generic message passing (i.e. throwing an exception object containing state rather than passing an object as a parameter).

In contrast, in cases where there is a single clear mode of failure that can be reasonably ignored then use a boolean.

Exception Test 1: Local (1)

```
public class ExceptionTest {
  private static void numberTest (int num) {
    boolean valid;

    try {
      if (num < 0 || num > 9)
        throw new RangeException("out of range, ");
      else
        valid = true;
    }
    catch (RangeException e) {
      System.out.print(e.getMessage());
      valid = false;
    }
```


Exception Test 1: Local (2)

```
    finally {
      System.out.print("always do this");
    }

    if (valid)
      System.out.print(", valid data");
  }
}
```

Testing Input:

When `main()` executes:

1. `numberTest(5);`
2. `numberTest(12);`
3. `numberTest(0);`
4. `numberTest(-1);`

Testing Output:

1. always do this, valid data
2. out of range, always do this
3. always do this, valid data
4. out of range, always do this

Exception Test 2: Hand Pass (1)

```
public class ExceptionTest2 {
  private static void numberTest (int num)
                        throws RangeException {
    if (num < 0 || num > 9)
      throw new RangeException("Number out of Range");
    else
      System.out.println("A valid number was entered");
  }

  public static void main(String args[]) {
    try {
      numberTest(5);
      numberTest(12);
      numberTest(0);
      numberTest(-1);
    }
```


Exception Test 2: Hand Pass (2)

```
    catch (RangeException e) {
      System.out.println(e.getMessage());
    }
    finally {
      System.out.println("Do this whatever happens");
    }
  }
}
```

Program Output:

A valid number was entered

Number out of range

Do this whatever happens

Exception Test 3: Do Nothing

```
public class ExceptionTest3 {

  private static void numberTest (int num)
                       throws RangeException {

    if (num < 0 || num > 9)
      throw new RangeException("Number out of Range");
    else
      System.out.println("A valid number was entered");
  }

  public static void main(String args[]) throws RangeException{

    numberTest(5);
    numberTest(12);
    numberTest(0);
    numberTest(-1);
  }
}
```

Program Output:

A valid number was entered

<crash; exception stack trace>

Exception Test 4: Combination (1)

```
public class ExceptionTest4 {
  private static int numberTest (int num, String s) {
    int convertedNum; int result = -1;
    try {
      if(num < 0 || num > 9)
          throw new RangeException("Number out of Range");
      convertedNum = Integer.parseInt(s);
      result = num / convertedNum;
    }
    catch( RangeException e) {
      System.out.println(e.getMessage());
    }
    catch( NumberFormatException e) {
      System.out.println(e.getMessage());
    }
    catch (ArithmeticException e) {
      System.out.println(e.getMessage());
    }
    return result;
  }
```

Here, a programmer defined exception is caught, and a library defined exception is passed on.

Where an exception is caught is a programmer's decision.
In general, handle an exception as near to the source as possible.

Exception Test 4: Combination (2)

```
  public static void main(String args[]) {
    int res;
    res = numberTest(5, "3");
    System.out.println(String.valueOf(res));
    System.out.println();
    res = numberTest(-1, "4");
    System.out.println(String.valueOf(res));
    System.out.println();
    res = numberTest(4, "sat");
    System.out.println(String.valueOf(res));
    System.out.println();
    res = numberTest(3, "0");
    System.out.println(String.valueOf(res));
    System.out.println();
  }
}
```


Exception Test 5: Variation on Test 4 (1)

Compare this with ExceptionTest 4.

```
public class ExceptionTest5 {
  private static int numberTest (int num, String s) {
    int convertedNum;
    int result = -1;
    try {
      if(num < 0 || num > 9)
          throw new RangeException("Out of Range");
      convertedNum = Integer.parseInt(s);
      result = num / convertedNum;
    }
```

Exception Test 5: Variation on Test 4 (2)

```
  catch(Exception e) {
    if (e instanceof RangeException)
      System.out.println(e.getMessage());
    else if (e instanceof NumberFormatException)
      System.out.println("Num. format exception");
    else if (e instanceof ArithmeticException)
      System.out.println("Arithmetic exception");
    else
      System.out.println("Unknown exception");
    }
  return result;
}
```


Java Exceptions and Repetition (1)

A common task is to repeatedly prompt the user for input until it is correct.

```
public class Adder {
  public static void main (String[] args) {
    int n1 = UserReader.getInt("Enter a number: ");
    int n2 = UserReader.getInt("Enter another number: ");

    System.out.println ("The sum is " + (num1+num2));
  }
}
```

The `UserReader` class is defined on the next slide.

Java Exceptions and Repetition (2)

```
class User_Reader {
  public static int getInt(String prompt) {
    BufferedReader stdin = new BufferedReader
                        (new InputStreamReader(System.in));
    int number = 0;
    boolean valid = false;

    while (! valid) {
      System.out.print (prompt);

      try {
        number = Integer.parseInt (stdin.readLine());
        valid = true;
      }
```


Java Exceptions and Repetition (3)

```
      catch (NumberFormatException exception) {
        System.out.println("Invalid input." +
                           "Try again.");
      }
      catch (IOException exception) {
        System.out.println ("Input problem." +
                            "Terminating.");
        System.exit(0);
      }

    }// end while loop

    return number;
  }
}
```

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 2: Inheritance and Polymorphism

Lecture Slides

Inheritance: the Credit Account Example

Account
acctNumber : int balance : double
withdraw(amount : double) : boolean deposit(amount : double) getAcctNumber() : integer getBalance() : double

The simple Account class from previously

The class has:

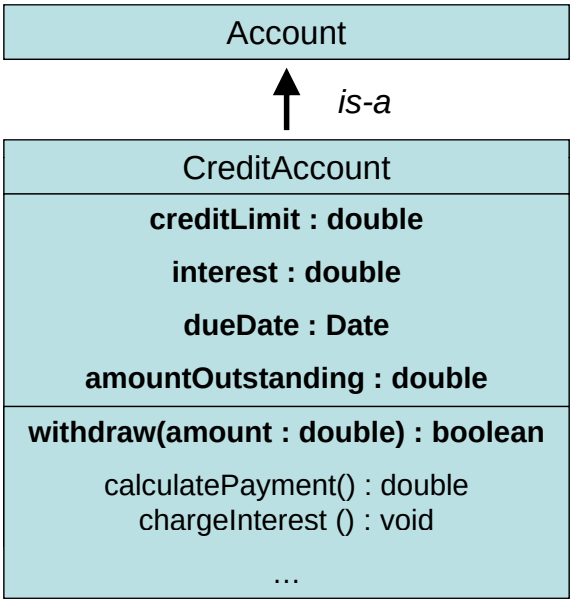
A *Name* (should be unique and descriptive)

Attributes (also called fields which are variables representing the **state** of the class)

Methods (which are functions representing the **behaviour** of the class)

Relationships to other classes represented as attributes (which may be collections for 'many' relationships)

Inheritance: the Credit Account Example



The bank introduces a new product, the credit account. This has a credit limit, an interest rate, a payment due date, and a payment amount outstanding.

A customer may withdraw up to the credit limit minus the account balance.

Every month, a credit charge based on the outstanding amount is be charged to the account.

Inheritance promotes **reusability** (code reuse) and **extensibility** (extending or modifying existing code).

Every object in Java, either explicitly or implicitly, is a subclass of `Object`.

The CreditAccount class: `CreditAccount.java` (1)

```
class CreditAccount extends Account {

  private double creditLimit = 0.0F;
  public double interest = 0.0F;
  private Date dueDate = new Date();
  private double outstanding = 0.0;

  public CreditAccount(int num, double creditLimit,
                       double interest)
  {
    super(num, 0); // set the balance to 0 for new account
    this.creditLimit = creditLimit;
    this.interest = interest;
  }

  public void deposit(double amount){
     setBalance(getBalance() - amount);
     outstanding -= amount;
     if (outstanding < 0)
       outstanding = 0;
  }
```

Inherits from `Account`

Calling the superclass constructor

Overriding the deposit method inherited from Account

The CreditAccount class: `CreditAccount.java` (2)

```
public boolean withdraw(double amount){
  if (creditLimit - getBalance() >= amount)
  {
    setBalance(getBalance() + amount);
    return true;
  }
  else
    return false;
}

public void chargeInterest(){
  setBalance(getBalance()+(getBalance() * (interest / 12)));
}
```

Overriding the withdraw method inherited from `Account`

A new method, `chargeInterest`

The CreditAccount class: `CreditAccount.java` (3)

```
public double calculateOutstanding(){
  if (getBalance() > 0)
      return getBalance() / interest; // not nec realistic!
  else
      return 0.0;
}
}
```

A new method, `calculateOutstanding`

In summary, the `CreditAccount` class:

- inherits properties from the `Account` class
- calls the superclass constructor to help create itself
- overrides some methods with a new implementation
- defines some new attributes and methods unique to `CreditAccount`

The super and this Keywords

A subclass may explicitly access a method or attribute in its superclass with the `super` keyword. When `super` is used to access the constructor of the superclass, it must be the first statement in the subclass' constructor.

The `this` keyword refers to the object through which the method or attribute is accessed (the current object.) In the absence of an object reference (or class name, for static items), this is the implied reference.

```
public CreditAccount(int num, double creditLimit,
                     double interest)
{
  super(num, 0); // set the balance to 0 for new account
  this.creditLimit = creditLimit; // this resolves the
  this.interest = interest;       // ambiguity
   balance=0; // this is not necessary since no ambiguity
}
```


Polymorphism and Dynamic Binding (1)

When the bank manager asks his/her staff to deduct $5.00 from every bank account, the staff know that different deduction methods will apply to different types of account. Hence the action deduction is polymorphic.

When the method `withdraw()` is called on an `Account` object, depending on the actual type of that `Account` at the time, the correct version of `withdraw()` will be dynamically bound to that object and executed.

First, let's add several accounts of different types for a customer:

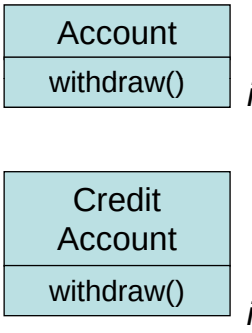
```
Customer john = new Customer("John", 123);
john.addAccount(new Account(1, 500));
john.addAccount(new CreditAccount(2, 0, 200));
```

Recall that the method `addAccount()` of `Customer` accepts an `Account` type as a parameter. Why can it also take in a `CreditAccount` type?

This is because `CreditAccount` "is-a" `Account`, due to inheritance.

Polymorphism and Dynamic Binding (2)

Now, to withdraw $5.00 from all of John's accounts. Which version of `withdraw()` is executed when?




```
Customer john = new Customer("John", 123);
john.addAccount(new Account(1, 500));
john.addAccount(new CreditAccount(2, 0, 200));

for (int i = 0; i < 2; i++) {

  Account a = john.getAccount(i);
  a.withdraw(5);
}
```

i=0

i=1

We can put different types of `Account` in a common `Account` array, and withdrawal can be performed polymorphically on them.

`Account` is the base class of all account subclasses. Polymorphism promotes code reuse by allowing methods to be called in a generic way.

Type Conversion

So, a `CreditAccount` can be treated as an `Account`. However, the reverse is not true; an `Account` may not be a `CreditAccount`.

```
Account ac;
CreditAccount creditAc;
ac = creditAc;
creditAc = ac;
```

Can we convert an `Account` to a `CreditAccount`? Yes, by **type casting**; this can be done only if the `Account` is actually a `CreditAccount`, otherwise a `ClassCastException` will be thrown.

```
creditAc = (CreditAccount) ac;
```

In summary:

- a subclass reference is automatically a superclass reference, but the reverse is not true.
- It is possible to convert a superclass reference to a subclass reference if the former is actually referencing a subclass object, otherwise Java will throw a `ClassCastException`.
- Avoid casting and use polymorphism to handle type differentiation

The `instanceof` Operator (1)

What if we call `chargeInterest()` on an `Account` object?

```
Customer john = new Customer("John", 123);
john.addAccount(new Account(1, 500));
john.addAccount(new CreditAccount(2, 0, 200));

for (int i = 0; i < 2; i++) {
  Account a = john.getAccount(i);
  a.withdraw(5);
  a.chargeInterest();
}
```

We know that we can call `chargeInterest()` on a `CreditAccount` object.

We also know that a superclass reference can be cast to a subclass reference, if it is actually holding a suitable subclass object.

The `instanceof` operator can perform a check to ensure this is the case.

The `instanceof` Operator (2)

```
Customer john = new Customer("John", 123);
john.addAccount(new Account(1, 500));
john.addAccount(new CreditAccount(2, 0, 200));

for (int i = 0; i < 2; i++) {
  Account a = john.getAccount(i);
  a.withdraw(5);
  if (a instanceof CreditAccount)
     ((CreditAccount) a).chargeInterest();
}
```

Checks if a is storing a `CreditAccount` object

An inline cast – a one-off shortcut

- **In general this technique should be avoided and polymorphism used instead**
- **Can use distinct collections and perform separate operations on them**
- **Characteristics unique to a class should be embedded within that class**

The `instanceof` Operator (3)

In summary, the `instanceof` operator is called in the following manner:

`<object reference> instanceof <class name>`

This will check whether or not the object referenced by <*object reference*> is an instance of class <*class name*>.

Additional Lecture Resources

PowerPoint slides from chapter 9 Inheritance and Polymorphism of the prescribed textbook *Introduction to Java Programming by Y. Daniel Liang, Prentice-Hall, 2007*, are available on Blackboard and will be presented during the lecture. You should study the book chapter and slides and may wish to print the slides so that you have a hard copy during the lecture.

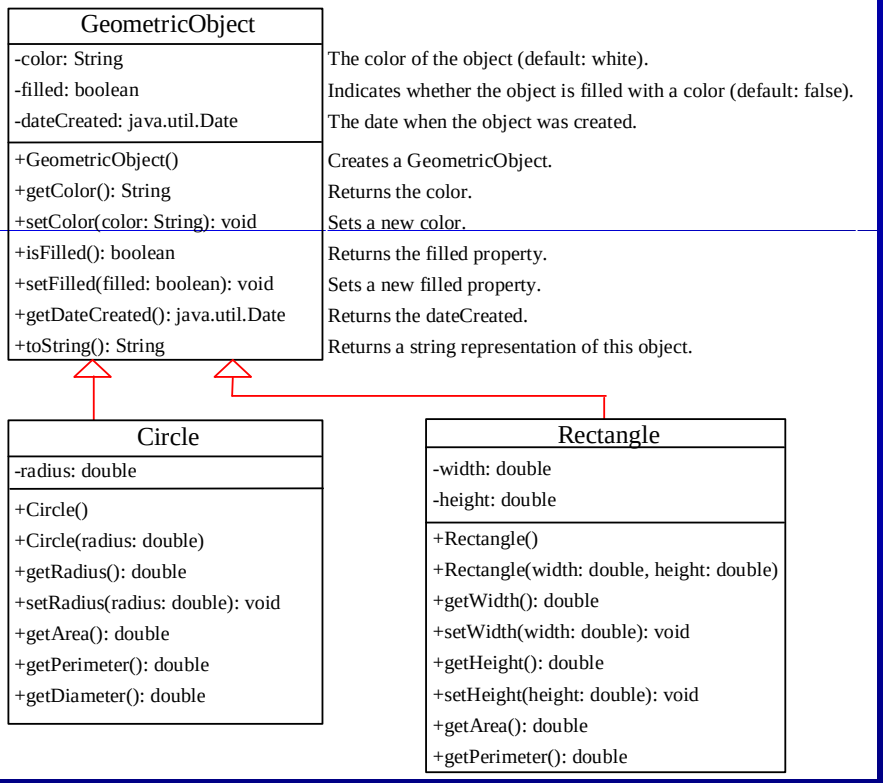
Chapter 9 Inheritance and Polymorphism

- These notes are a subset, for use in cosc1076 Programming 2, of the chapter 9 lecture notes from the prescribed textbook: Introduction to Java Programming by Y. Daniel Liang, Prentice-Hall, 2007

Objectives

- To develop a subclass from a superclass through inheritance (§9.2).
- To invoke the superclass's constructors and methods using the super keyword (§9.3).
- To override methods in the subclass (§9.4).
- To distinguish differences between overriding and overloading (§9.5).
- To comprehend polymorphism, dynamic binding, and generic programming (§9.7).
- To describe casting and explain why explicit downcasting is necessary (§9.8).
- To restrict access to data and methods using the protected visibility modifier (§9.11).
- To declare constants, unmodifiable methods, and nonextendable classes using the final modifier (§9.12).

Superclasses and Subclasses



GeometricObject

Circle

Rectangle

TestCircleRectangle

Run

Are superclass's Constructor Inherited?

No. They are not inherited.

They are invoked explicitly or implicitly.

Explicitly using the super keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```


Using the Keyword super

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is called *constructor chaining*.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Declaring a Subclass

A subclass extends properties and methods from the superclass. You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass

Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
public class Circle extends GeometricObject {
  // Other methods are omitted


  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```


NOTE

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

NOTE

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

The `Object` Class

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```


The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5 . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Polymorphism Demo

Run

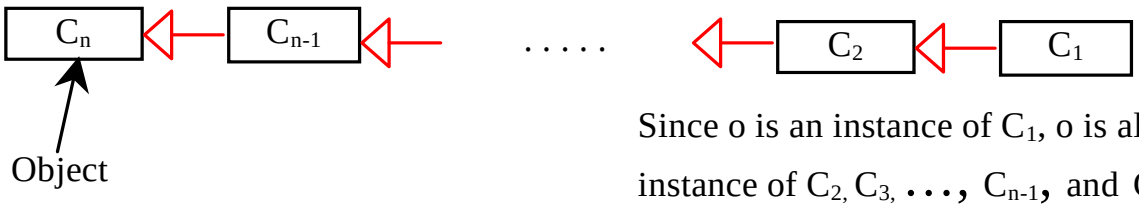
Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Since o is an instance of C_1, o is also an instance of C_2, C_3, . . . , C_{n-1}, and C_n

Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime. See Review Questions 9.7 and 9.9.

Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compilation error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

Use the `instanceof` operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of
  Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

TIP

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

Example: Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects. The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

TestPolymorphismCasting

Run

The `protected` Modifier

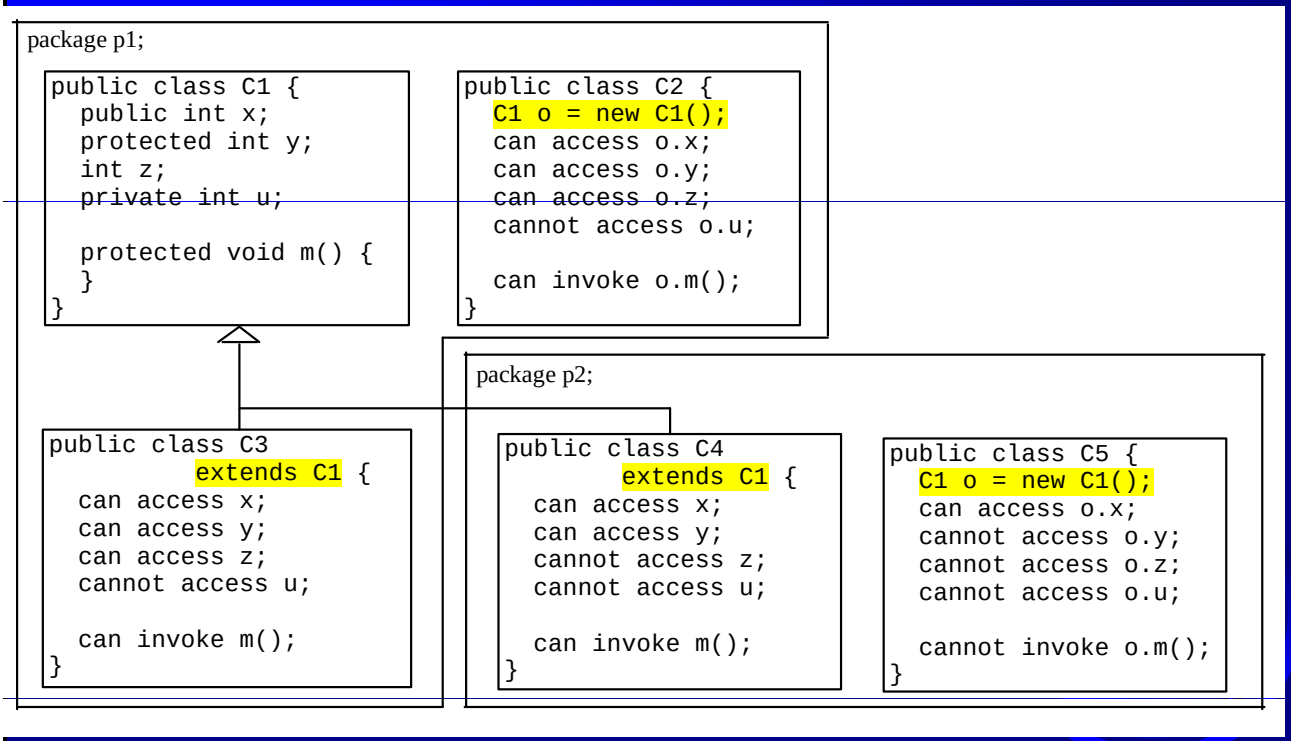
- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public

```
Visibility increases
——————————————————→
private, none (if no modifier is used), protected, public
```


Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

Visibility Modifiers



A Subclass Cannot Weaken the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

NOTE

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

Object.equals() versus ==

The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 3: Abstract Classes and Interfaces

Lecture Slides

What is an `abstract` class?

Simply put, an abstract class is a Java class **that has not been fully defined.** For example, it may have the implementation of a method missing.

This can be useful, as it allows the subclasses to inherit and override these abstract methods; and it gives each of these subclasses **a consistent programming interface.**

Furthermore, the “gap” that the abstract class fills in the class hierarchy **can be used as a declaration type** to handle any extension of the abstract class.

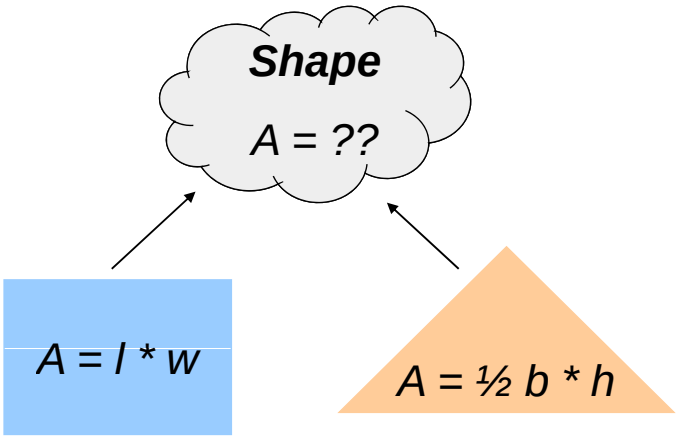
Note: a **fully implemented** (normal) class is referred to as a **concrete class.**

Abstract Classes: an Example

Take the example of a series of shapes, that is to be modelled in a Java class hierarchy.

All shapes will have a colour, and other common elements, however (for example) the manner in which the area of the shape is calculated would depend on what kind of shape it is.

Therefore, it would make conceptual sense to give the area method an implementation only in the concrete subclasses; we can make it abstract to ensure this.



Implementing an Abstract Class

An abstract class (or an abstract method) is defined using the keyword **`abstract`**. It is also possible to define an abstract class that has no abstract properties!

```
abstract class Shape {
  protected Color col;

  public abstract double area();
}
```

Note the lack of an implementation for the **`area()`** method, and the need to declare the class as abstract.

Extending an Abstract Class

An extension of an abstract class only becomes concrete if there are no abstract methods left (otherwise it must itself be declared abstract). In our `Rectangle` class we must therefore override the **`area()`** method:

```
class Rectangle extends Shape {
  double length, width;

  public double area() {
    return (length * width);
  }
}
```


Using Abstract Classes

If there were any `abstract` properties left in our subclass of `Shape`, that subclass could be defined as `abstract`, and a compiler error avoided.

However, it is not possible to make an instance of (instantiate using new) an abstract class. In order to make instances of a class, it must be fully defined i.e. Be a non-abstract concrete class.

```
Shape shape = new Shape();           // this will not work
Rectangle shape2 = new Rectangle();  // this is allowed
```

With the class hierarchy having the abstract **Shape** as its parent, it allows properties common to all Shapes (such as the colour) to be defined and inherited, thus re-using code as much as possible; but it still allows us to leave some implementation details for further down the class hierarchy.

Abstract Classes and Polymorphism (1)

Although abstract classes lack some implementation details, they can still be used in polymorphic behaviour.

```
Shape shape = new Rectangle();
shape.length = 2.5;
shape.width = 5.0;

System.out.println(shape.area());
```

Note that since `shape` is of the static (compile-time) type **`Shape`** we can only call methods declared inside **`Shape`** not specialised methods in **`Rectangle`**.

Abstract Classes and Polymorphism (2)

Here is another example; it is a method that will take any **`Shape`** object, and print its area.

```
public void printArea(Shape in) {
  System.out.println("Area: " + in.area());
}
```

Recall earlier that we could not directly **create** an object from the abstract **`Shape`** class. However in these cases, we are working on concrete subclasses of **`Shape`**; we are just using the **`Shape`** data type as a handle to refer to any subclass.

Remember the ‘is-a’ relationship: *“a **`Rectangle`** is a **`Shape`**”*

What is an interface?

An interface is a **definition of a programming contract** that classes can **implement**. It is **not** a class and cannot be instantiated.

It is possible for a class to implement many interfaces.

The major benefit of using an interface is that it allows otherwise unrelated classes to be given a common **type** without the need for multiple inheritance (which Java forbids.)

If a class implements a particular interface, it must **directly implement** all of the methods of that interface or it must be declared abstract.

If a class that implements an interface is extended, **those subclasses also implement the interface**. (This is because the methods that make the implementation will be inherited.)

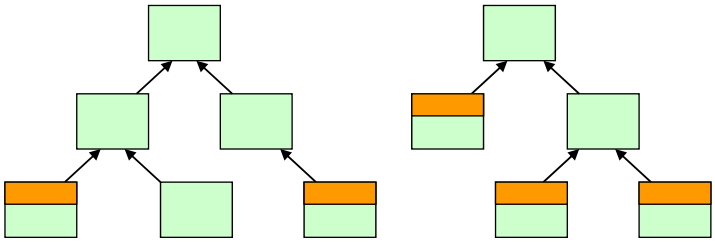
Interfaces: an Example

Consider a set of class hierarchies, some of whose objects can be printed out to the screen.

We could **create an interface** defining how the “printable” functionality is to be implemented, and have these classes implement that interface.

The benefit is that we can then use that interface **as a data type**, much like we can with superclasses and polymorphism.

All `Printable` classes in the diagram can be referred to by their capability, even though they are completely unrelated in the hierarchy.



Defining an Interface

The code to define an `interface` is somewhat similar to an `abstract` class; no method implementations are given, and use of the **`abstract`** keyword is optional. Again, remember that an interface is not a class and cannot be instantiated.

```
interface Printable {
  public void print();
}
```

We can also extend an interface to make a larger sub-interface. This is done using the **`extends`** keyword, much like extending a class. The new interface has the methods of the interface it extends plus any new methods it declares.

Implementing an Interface

In order to get a class to implement the interface, we use the **`implements`** keyword:

```
class StringData implements Printable {
  private String data;

  public void print() {
    System.out.println("Data: " + data);
  }
}
```

We **must** either provide an implementation for **`print()`** or declare the StringData class abstract.

An Interface as a Data Type

As with abstract classes, we can use our **`Printable`** interface as a data type:

```
public void printObject(Printable in) {
  in.print();
}
```

This method will accept any object that implements **`Printable`**, and call its **`print()`** method.

However, when referring to the objects as **`Printable`**, we can only call what is defined inside the **`Printable`** interface.

Using interfaces allows us to write code that can operate on a wide range of specialised data types (by using interface types to declare variables, as parameter types etc.)

An Example: the Shape class

Let's write a more formal example of abstract classes and interfaces.

Specification: define a set of classes to process geometric shapes. There is no basic shape from which all shapes derive, so we place an abstract class at the root of the hierarchy.

```
abstract class Shape {
  public abstract double area();
  public abstract double perimeter(); // yes this is ok!
}
```

The next step is to extend Shape to handle circles and rectangles, and then create instances of these classes:

```
Shape aRect = new Rectangle();
Shape aCircle = new Circle(3.0);
```

An Example: the `Circle` class

```
class Circle extends Shape {
  protected double r;

  Circle(double r) {
    this.r = r;
  }

  public double area() {
    return Math.PI * r * r;
  }

  public double perimeter() {
    return 2 * Math.PI * r;
  }
}
```


An Example: the `Rectangle` class

```
class Rectangle extends Shape
{
   protected double w, h;

   Rectangle(double w, double h)
   {
      this.w = w;    this.h = h;
   }

   public double area()
   {
      return w * h;
   }

   public double perimeter()
   {
      return 2 * (w + h);
   }
}
```

Introducing Interfaces

Suppose we want to derive a subclass of `Rectangle` that allows a rectangle to be drawn on a window object. We might also want to do this with circles as well. What can we do?

Rewrite the Shape hierarchy to include drawing methods

We would override these in each non-abstract shape.

But what if not all shapes are Drawable?

Define drawing methods in each non-abstract drawable shape?

Loses polymorphic behaviour

Can we define a class that extends `Shape` but also has a 'parent' of sorts, that provides methods for drawing shapes in a window object? Java does not allow multiple inheritance, but we can use an `interface` for this job.

```
interface Drawable {
  public void setColor(Color c);
  public void setPosition(double x, double y);
}
```


The `DrawableRectangle` class

```
class DrawableRectangle extends Rectangle implements Drawable {

   private Color c;
   private double x, y;

   DrawableRectangle(double w, double h) {
     super.w = w;
     super.h = h;
   }

   public void setColor(Color c) {
     this.c = c;
   }

   public void setPosition(double x, double y) {
     this.x = x;
     this.y = y;
   }
}
```

Design with Abstract Classes and Interfaces

Why abstract classes?

We want a generalised base class that is incomplete (and thus cannot be accidentally instantiated) but want to provide default behaviour (that may be extended or specialised later).

Why interfaces?

- We want to treat different classes together in some specific way by means of a strict contractual interface (set of method signatures) but have no default behaviour
- We have many such behaviours and therefore the need for multiple inheritance which is only supported at the interface level in Java.

Note that like regular inheritance abstract classes and interfaces ensure a uniform and structured design throughout the class hierarchy. Abstract classes, inheritance and interfaces, together with polymorphism and dynamic binding, promote code reuse by allowing methods to be called in a generalised way. It also facilitates maintainability by placing specialised code in a single place.

Abstract Classes vs. Interfaces

A side-by-side comparison of the two:

Abstract Class	Interface
1. Used by extending.	1. Used by implementing.
2. Can have `abstract` and concrete methods.	2. Can have only `abstract` methods.
3. Need not implement all abstract methods.	3. No methods are implemented.
4. Can have any modifiers on data types.	4. Data must be `public static final` constants.
5. A class cannot extend multiple abstract classes.	5. A class can implement many interfaces.
6. Cannot be instantiated.	6. Cannot be instantiated.
7. Can be used as a data type.	7. Can be used as a data type.

Additional Lecture Resources

PowerPoint slides from chapter 10 Abstract Classes and Interfaces of the prescribed textbook *Introduction to Java Programming by Y. Daniel Liang, Prentice-Hall, 2007*, are available on Blackboard and will be presented during the lecture. You should study the book chapter and slides and may wish to print the slides so that you have a hard copy during the lecture.

Chapter 10 Abstract Classes and Interfaces

- These notes are a subset, for use in cosc1076 Programming 2, of the chapter 10 lecture notes from the prescribed textbook: Introduction to Java Programming by Y. Daniel Liang, Prentice-Hall, 2007

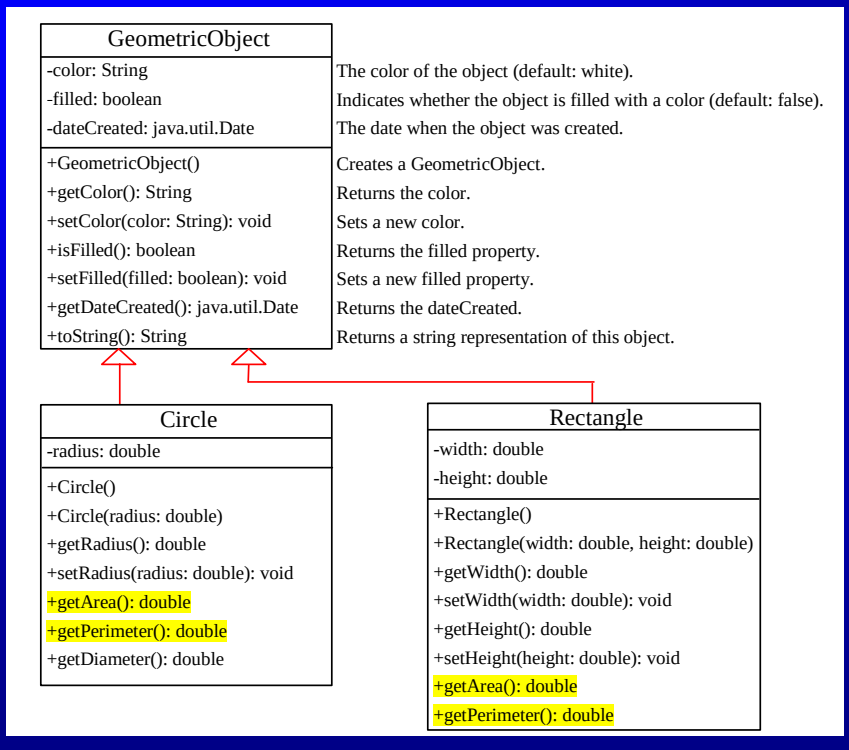
Objectives

- To design and use abstract classes (§10.2).
- To declare interfaces to model weak inheritance relationships (§10.4).
- To know the similarities and differences between an abstract class and an interface (§10.4.2).
- To declare custom interfaces (§10.4.3).
- To use wrapper classes (Byte, Short, Integer, Long, Float, Double, Character, and Boolean) to wrap primitive data values into objects (§10.5).
- To simplify programming using JDK 1.5 automatic conversion between primitive types and wrapper class types (§10.6).

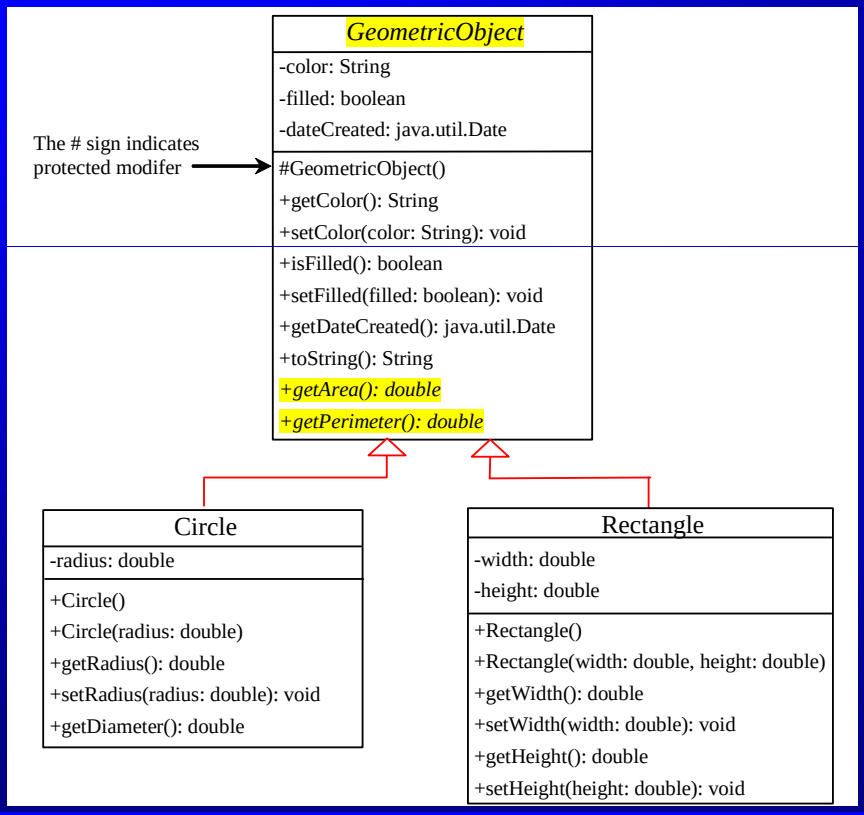
The `abstract` Modifier

- The `abstract` class
  - Cannot be instantiated
  - Should be extended and implemented in subclasses
- The `abstract` method
  - Method signature without implementation

From Chapter 9



Abstract Classes



GeometricObject

Circle

Rectangle

NOTE

An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be declared abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

NOTE

An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

NOTE

A class that contains abstract methods must be abstract. However, it is possible to declare an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

NOTE

A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

NOTE

A subclass can override a method from its superclass to declare it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be declared abstract.

NOTE

You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new
GeometricObject[10];
```

Example: Using the GeometricObject Class

- Objective: This example creates two geometric objects: a circle, and a rectangle, invokes the equalArea method to check if the two objects have equal area, and invokes the displayGeometricObject method to display the objects.

TestGeometricObject

Run

Interfaces

An *interface* is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain variables and concrete methods as well as constants and abstract methods.

To distinguish an interface from a class, Java uses the following syntax to declare an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```


Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

Example Using Interfaces

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

```
class Animal {
}

class Chicken extends Animal
    implements Edible {
  public String howToEat() {
    return "Fry it";
  }
}

class Tiger extends Animal {
}
```

```
class abstract Fruit
    implements Edible {
}
```

```
class Apple extends Fruit {
  public String howToEat() {
    return "Make apple cider";
  }
}

class Orange extends Fruit {
  public String howToEat() {
    return "Make orange juice";
  }
}
```


Implements Multiple Interfaces

```
class Chicken extends Animal implements Edible, Comparable {
  int weight;
  public Chicken(int weight) {
    this.weight = weight;
  }
  public String howToEat() {
    return "Fry it";
  }
  public int compareTo(Object o) {
    return weight – ((Chicken)o).weight;
  }
}
```

Creating Custom Interfaces, cont.

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

```
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++)
      showObject(objects[i]);
  }

  public static void showObject(Object object) {
    if (object instanceof Edible)
      System.out.println(((Edible)object).howToEat());
  }
}
```


Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

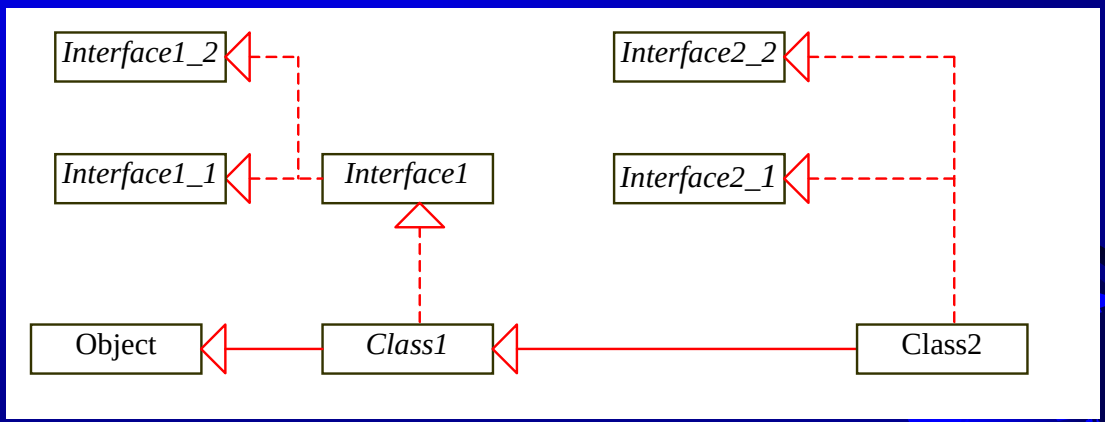
Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If an interface extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Caution: conflict interfaces

In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type). This type of errors will be detected by the compiler.

Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person. So their relationship should be modeled using class inheritance. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface. See Chapter 10, "Object-Oriented Modeling," for more discussions.

General Purpose Interfaces

Suppose you want to design a generic method to find the larger of two objects. The objects can be students, dates, or circles. Since compare methods are different for different types of objects, you need to define a generic compare method to determine the order of the two objects. Then you can tailor the method to compare students, dates, or circles. For example, you can use student ID as the key for comparing students, radius as the key for comparing circles, and volume as the key for comparing dates. You can use an interface to define a generic compareTo method, as follows:

Example of an API Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable {
  public int compareTo(Object o);
}
```

String and Date Classes

Many classes (e.g., String and Date) in the Java library implement Comparable to define a natural order for the objects. If you examine the source code of these classes, you will see the keyword implements used in the classes, as shown below:

```
public class String extends Object
    implements Comparable {
  // class body omitted
}
```

```
public class Date extends Object
    implements Comparable {
  // class body omitted
}
```

```
new String() instanceof String
new String() instanceof Comparable
new java.util.Date() instanceof java.util.Date
new java.util.Date() instanceof Comparable
```


Generic `max` Method

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Comparable max
      (Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(a)

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Object max
      (Object o1, Object o2) {
    if ((Comparable)o1).compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(b)

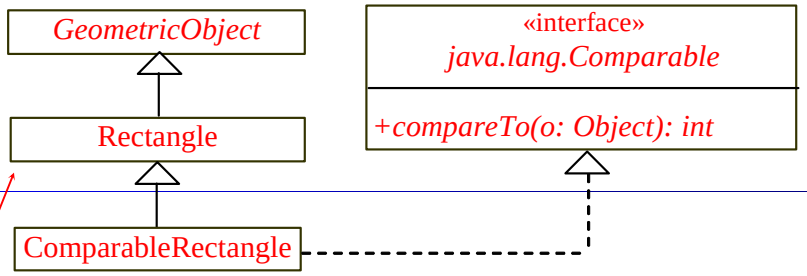
```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

The return value from the max method is of the Comparable type. So, you need to cast it to String or Date explicitly.

Declaring Classes to Implement Comparable

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.



ComparableRectangle

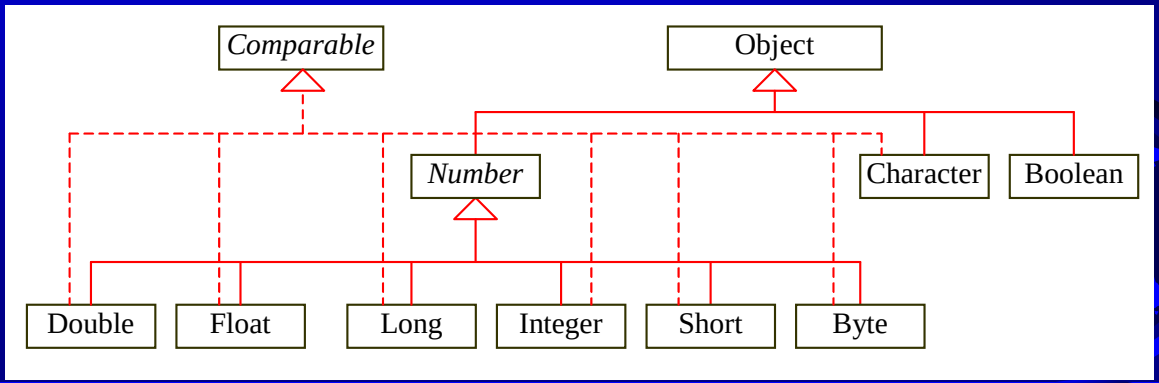
You cannot use the max method to find the larger of two instances of Rectangle, because Rectangle does not implement Comparable. However, you can declare a new rectangle class that implements Comparable. The instances of this new class are comparable. Let this new class be named ComparableRectangle.

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```


Wrapper Classes

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double

NOTE: (1) The wrapper classes do not have no-arg constructors. (2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.



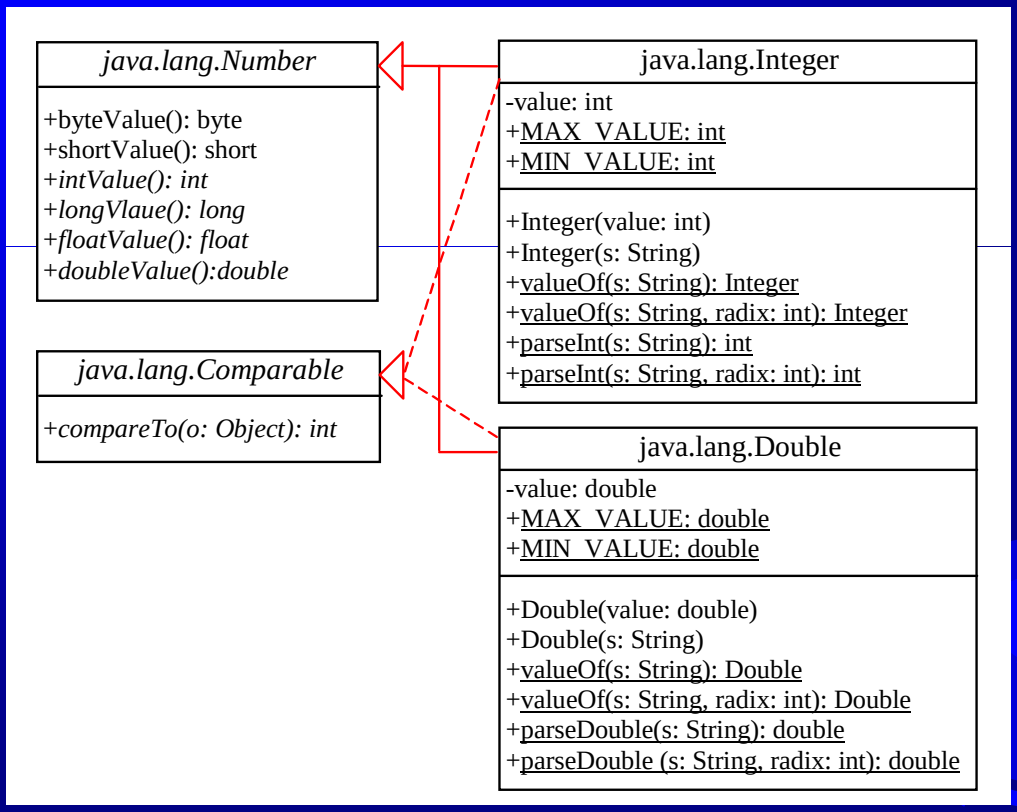
The toString, equals, and hashCode Methods

Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class. Since all the numeric wrapper classes and the Character class implement the Comparable interface, the compareTo method is implemented in these classes.

The Number Class

Each numeric wrapper class extends the abstract Number class, which contains the methods doubleValue, floatValue, intValue, longValue, shortValue, and byteValue. These methods "convert" objects into primitive type values. The methods doubleValue, floatValue, intValue, longValue are abstract. The methods byteValue and shortValue are not abstract, which simply return (byte)intValue() and (short)intValue(), respectively.

The `Integer` and `Double` Classes



The `Integer` Class and the `Double` Class

- Constructors
- Class Constants `MAX_VALUE`, `MIN_VALUE`
- Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE represents the minimum *positive* float and double values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).

Conversion Methods

Each numeric wrapper class implements the abstract methods doubleValue, floatValue, intValue, longValue, and shortValue, which are defined in the Number class. These methods "convert" objects into primitive type values.

The Static valueOf Methods

The numeric wrapper classes have a useful class method, valueOf(String s). This method creates a new object initialized to the value represented by the specified string. For example:

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

The Methods for Parsing Strings into Numbers

You have used the parseInt method in the Integer class to parse a numeric string into an int value and the parseDouble method in the Double class to parse a numeric string into a double value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

JDK 1.5 Feature

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):

```
Integer[] intArray = {new Integer(2),
  new Integer(4), new Integer(3)};
```

(a)

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

(b)

New JDK 1.5 boxing

```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 4: Java Collection Framework and Generics/Parameterized Typing

Lecture Slides

Java Collections

- An Object that groups multiple elements into a single unit
- To store, retrieve and manipulate data, and to transmit data from one method to another
- Typically represents data items belonging to a natural group, such as Library Catalogue (a collection of library materials and borrow-status)

JAVA Collection Implementation in earlier versions (still supported)

1. Vector - collection of Java Objects (contents can be instantiated from different classes) without prior knowledge of the size
2. Hashtable – collection of name-value pairs, like a dictionary, easy lookup
3. Array – collection of objects of same class or same primitive datatypes with known size

Java Collection Framework (JCF)

A 'Unified Architecture' for representing and manipulating collections

Collection Framework contains:

• **Interfaces**: abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation.

`java.util.Collection etc.`

• **Implementations**: concrete implementations of the collection interfaces. In essence, these are *reusable data structures*.

`java.util.ArrayList etc.`

• **Algorithms**: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces. These algorithms are said to be *polymorphic* because the same method can be used on many different implementations of the appropriate collections interface. In essence, algorithms are *reusable functionality*.

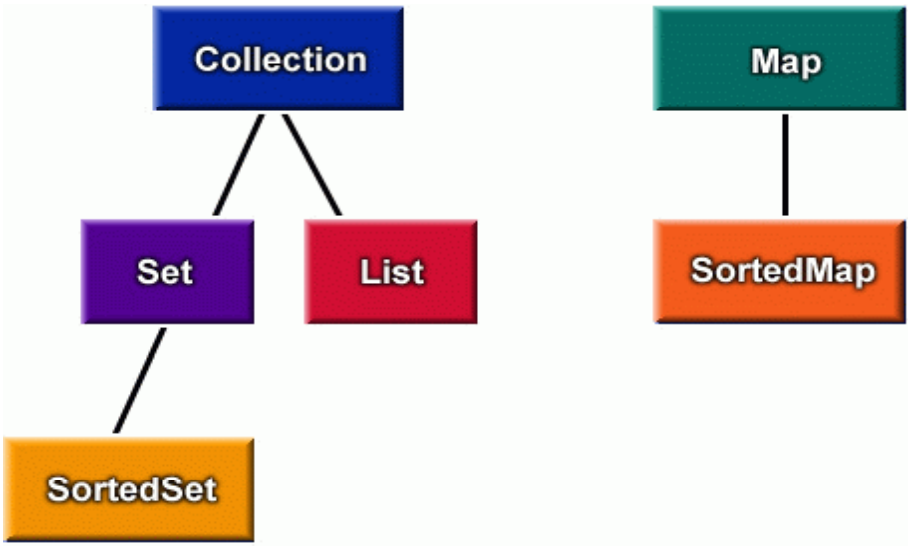
`java.util.Collections` (note the 's' on the end!)

One of the basic features of OO Programming is *reusability*.

Pros and Cons of JCF

Pros	Cons
1. Reduces Programming Effort (?)	1. Complexity (?)
2. Increases Speed and Quality (?)	
3. Allows interoperability among unrelated code	
4. Reduces learning effort (?)	
5. Reduces design effort	
6. Fosters software reuse	

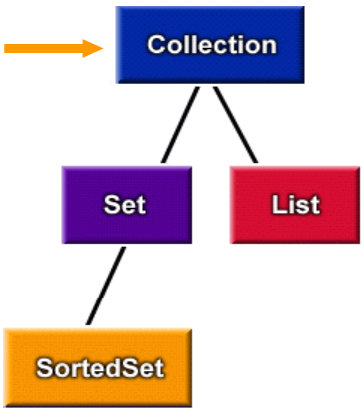
Collection Interface Hierarchy



No unifying interface but can retrieve a Collection from a Map with Map.values()

Modification operations in each interfaces are designated *optional*: a given implementation may not support some of these operations. If an unsupported operation is invoked, a collection throws an UnsupportedOperationException

Interface : Collection



What is Iterator?

A Collection represents a group of objects, known as its *elements*. The primary use of the Collection interface is to pass around collections of objects where maximum generality is desired.

```
public interface Collection
{
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Iterator

Iterator Interface (new)

Iterator allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

```
public interface Iterator
{
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```

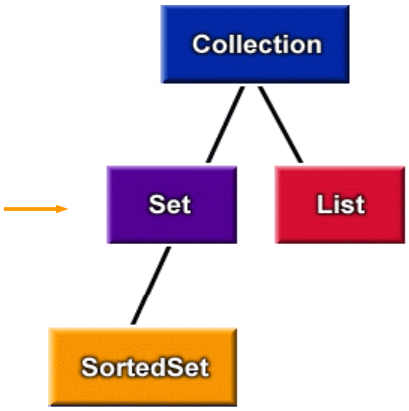
Enumeration Interface (old)

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series. Eg, StringTokenizer

Two Methods:

- `boolean hasMoreElements()`
- `Object nextElement()`

Interface : Set



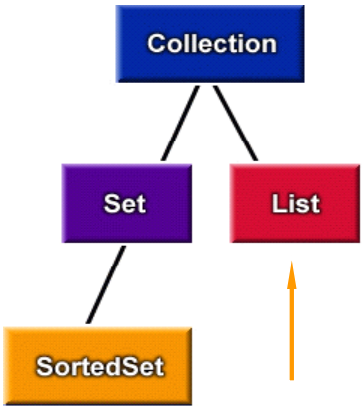
- A **Set** is a Collection that cannot contain duplicate elements.
- Set models the mathematical *set* abstraction.
- The Set interface extends Collection and contains *no* methods other than those inherited from Collection.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations (inherited from Object), allowing Set objects with different implementation types to be compared meaningfully. Two Set objects are equal if they contain the same elements.

JDK Standard Implementation
HashSet – best performing and stored in hash table.
TreeSet - guarantees ordering and stored in red-black tree

What are the common Set Operations?

Subset	`s1.containsAll(s2)`
Union	`s1.addAll(s2)`
Intersection	`s1.retainAll(s2)`
Set Difference	`s1.removeAll(s2)`

Interface: List



A **List** is an ordered Collection (sometimes called a *sequence*). Lists may contain duplicate elements.

In addition to the operations inherited from Collection, the List interface includes operations for:

1. **Positional Access (indexed)**: manipulate elements based on their numerical position in the list.
2. **Search**: search for a specified object in the list and return its numerical position.
3. **List Iteration**: extend Iterator semantics to take advantage of the list's sequential nature.
4. **Range-view**: perform arbitrary *range operations* on the list.

JDK Standard Implementation

ArrayList – best performing and stored in hash table.
LinkedList - better performance under certain circumstances
Vector – retrofitted to implement List

Interface: List (Continued…)

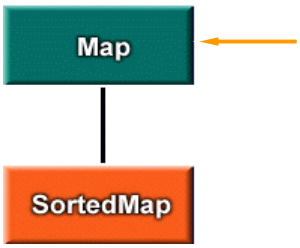
```
public interface List extends Collection {
   // Positional Access
   Object get(int index);
   Object set(int index, Object element); // Optional
   void add(int index, Object element);   // Optional
   Object remove(int index);              // Optional
   abstract boolean addAll(int index,
                           Collection c); // Optional

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator listIterator();
   ListIterator listIterator(int index);

   // Range-view
   List subList(int from, int to);
}
```

```
public interface ListIterator
   extends Iterator {

   boolean hasNext();
   Object next();

   boolean hasPrevious();
   Object previous();

   int nextIndex();
   int previousIndex();

   void remove();          // Optional
   void set(Object o);     // Optional
   void add(Object o);     // Optional
}
```

Most of the polymorphic algorithms in the **JCF** classes (e.g. java.util.Collections) apply specifically to List not Collection.

- `sort(List)`
- `shuffle(List)`
- `reverse(List)`
- `fill(List, Object)`
- `copy(List dest, List src)`
- `binarySearch(List, Object)`

Interface: Map



A Map is a collection that maps keys to values.

A map cannot contain duplicate keys: Each key can map to at most one value.

JDK Standard Implementation
HashMap– best performing and stored in hash table.
TreeMap - guarantees ordering and stored in red-black tree
Hashtable – retrofitted to implement Map

```
public interface Map
{
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry
    {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```


Comparable and Comparator Interface

If the list consists of String elements, it will be sorted into lexicographic (alphabetical) order. If it consists of Date elements, it will be sorted into chronological order. How?

Both String and Date implements the ***Comparable*** interface that provides a ***natural ordering*** for a class, which allows objects of that class to be sorted automatically.

- a list whose elements do not implement Comparable, Collections.sort(list) will throw a ClassCastException.

- a list whose elements cannot be compared *to one another*, Collections.sort will throw a ClassCastException.

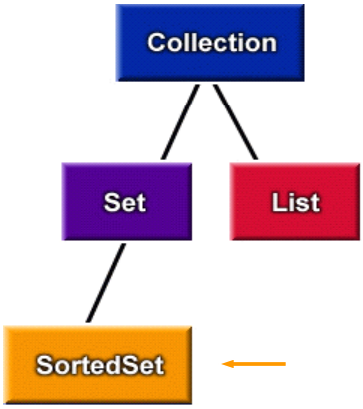
- Elements that can be compared to one another are called *mutually comparable*. While it is possible to have elements of different types be mutually comparable, none of the JDK types (Byte, Character, Long, Integer, Short, Double, Float, String, Date) permit inter-class comparison.

```
public interface Comparable
{
   public int compareTo(Object o);
}
```

```
public interface Comparator
{
   int compare(Object o1, Object o2);
}
```

Interface: SortedSet

A **SortedSet** is a Set that maintains its elements in ascending order, sorted according to the elements' ***natural order***, or according to a Comparator provided at SortedSet creation time.



In addition to the normal Set operations, the SortedSet interface provides operations for:

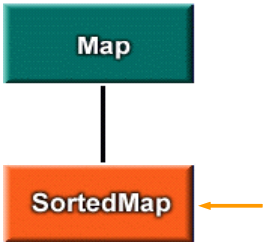
Range-view: Performs arbitrary *range operations* on the sorted set.
Endpoints: Returns the first or last element in the sorted set.
Comparator access: Returns the Comparator used to sort the set

```
public interface SortedSet extends Set {
   // Range-view
   SortedSet subSet(Object fromElement, Object toElement);
   SortedSet headSet(Object toElement);
   SortedSet tailSet(Object fromElement);

   // Endpoints
   Object first();
   Object last();

   // Comparator access
   Comparator comparator();
}
```


Interface: SortedMap

A **SortedMap** is a Map that maintains its elements in ascending order, sorted according to the elements' ***natural order***, or according to a Comparator provided at SortedMap creation time.

In addition to the normal Map operations, the SortedMap interface provides operations for:



Range-view: Performs arbitrary *range operations* on the sorted map.
Endpoints: Returns the first or last key in the sorted map.
Comparator access: Returns the Comparator used to sort the map

```
public interface SortedMap extends Map {
    // Range-view
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    // Endpoints
    Object first();
    Object last();

   // Comparator access
    Comparator comparator();

}
```

Implementation

Implementations are the actual data objects used to store collections, which implement the *core collection interfaces*

General-purpose Implementations

General-purpose implementations are the public classes that provide the primary implementations of the core collection interfaces. e.g. ArrayList, HashMap

Convenience Implementations

Convenience implementations are mini-implementations, typically made available via *static factory methods* that provide convenient, efficient alternatives to the general-purpose implementations for special collections e.g. Collections.singletonList().

Wrapper Implementations

Wrapper implementations are used in combination with other implementations (often the general-purpose implementations) to provide added functionality. e.g. Collections.unmodifiableCollection();

General Purpose Implementations

Two implementations for each interface except Collection has been provided.

		Implementations			
		Hash Table	***Resizable Array***	***Balanced Tree***	***Linked List***
Interfaces	***Set***	***HashSet***		***TreeSet***	
	List		***ArrayList, Vector***		***LinkedList***
	Map	***HashMap***		***TreeMap***	

(Reference : Java Tutorial – Joshua Bloch, collection trail)

An Iterator Example

```
c1 = new LinkedList();
// Process the linked list
Iterator iter = c1.iterator();
while (iter.hasNext()) {
   Object obj = iter.next();
   //do what you want to obj
}
c2 = new HashSet();
// Process the hash set
iter = c2.iterator();
while (iter.hasNext()) {
   Object obj = iter.next();
   //do what you want to obj
}
```

Note identical code for generating the sequence of objects

typically, we must cast the object to use it properly

Java 1.5: Use of for-each loop instead of an Iterator

```
c1 = new LinkedList();
// Process the linked list
Iterator iter = c1.iterator();
while (iter.hasNext()) {
   Object obj1 = iter.next();
   //do what you want to obj1
}
c2 = new HashSet();
// Process the hash set
iter = c2.iterator();
while (iter.hasNext()) {
   Object obj2 = iter.next();
   //do what you want to obj2
}
```

```
c1 = new LinkedList();
// Process the linked list
for(Object obj1: c1)
   //do what you want to obj1
}

c2 = new HashSet();
// Process the hash set
for(Object obj2: c2) {
   //do what you want to obj2
}
```

Use of Generic classes

- Java 5 allows the use of Generic classes and methods
- Traditionally Java programmers have used inheritance & polymorphism to create flexible classes.
- A flexible Stack class can be written to store any object by creating an array of Object references
- However it does not prevent the wrong type of object being added to the stack at compile time – resulting in a runtime error at a later time.
- It also requires casting when an object is retrieved.

```
Class Stack
{
   private Object elems[];
   void push(Object o) {…}
   Object pop() {…}
}

Stack custStack = new Stack(10);

custStack.push(new Customer(…));
custStack.push(new Customer(…));
custStack.push(new Account(…));
…
Cutomer c = (Customer)
custStack.pop();
…
```


Using JCF Generic classes

- All the JCF classes we have seen before are created to be used as generic classes
- For example, an ArrayList instance can be created to store only Account objects by passing the type (Account) to the constructor and the reference as in:

```
List<Account> accList = new ArrayList<Account>();
```

- Similarly to map a customer name (String) to Account objects we can use:

```
Map<String,Account> hashMap = new
   HashMap<String,Account>();
```

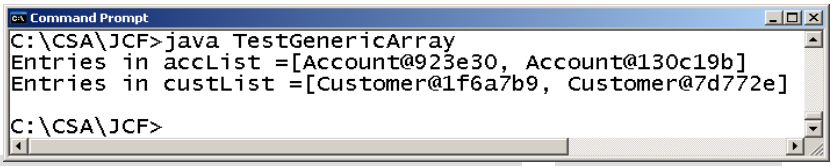
- The same classes when used without specifying any type information reduces to raw type which is equivalent to:

```
List<Object> accList = new ArrayList<Object>();
Map<Object,Object> hashMap = new
   HashMap<Object,Object>();
```

Example using ArrayList

```
import java.util.*;
class Account {  }
class Customer { }

public class TestGenericArray {
   public static void main(String[] args) {
      List<Account> accList = new ArrayList<Account>();
      accList.add( new Account() );
      accList.add( new Account() );

      List<Customer> custList = new ArrayList<Customer>();
      custList.add( new Customer() );
      custList.add( new Customer() );
//    custList.add( new Account() );

      System.out.println("Entries in accList =" + accList);
      System.out.println("Entries in custList =" + custList);

   }
}
```

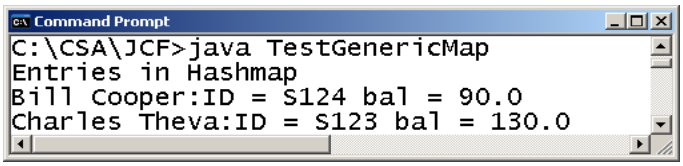
compiler detects type mismatch



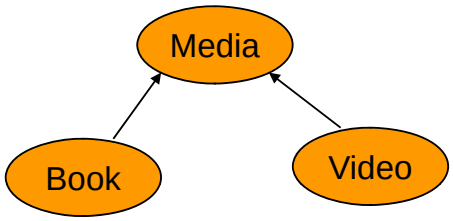
Example using HashMap

```
import java.util.*;
Customer Account { …
public class TestGenericMap {
  public static void main(String[] args) {
  HashMap<String,Account> hashMap =
                  new HashMap<String,Account>();
      hashMap.put("Charles Theva", new Account("S123",130.0));
      hashMap.put("Bill Cooper", new Account("S124",90.0));
//    hashMap.put(1234, new Account("S126",220.0));
      System.out.println("Entries in Hashmap");
      displayMap(hashMap);
   }
   public static void displayMap(Map<Integer, Account> m)
   {
      Set<Integer> keySet = m.keySet();
      Iterator<Integer> iterator = keySet.iterator();
      while (iterator.hasNext())
      {
         Integer key = iterator.next();
         System.out.println(key + ":“
                 + m.get(key));
      }
   }
}
```

Compiler detects type mismatch



Consider a simple library system




```
import java.util.*;
class Library {
   private List resources =
             new ArrayList();
   public void add(Media x) {
      resources.add(x);
   }
   public Media retrieveLast() {
      int size = resources.size();
      if (size > 0) {
         return (Media)
           resources.get(size-1);
      }
      return null;
   }
}
```

```
public class TestLibrary
{  public static void main(String  args[])
   {  Library myBooks =  new Library();
      myBooks.add(new Book());
      myBooks.add(new Book());
//    myBooks.add(new Video());
      Book lastBook = (Book) myBooks.retrieveLast();
      lastBook.print();
   }
}
```

For storing Book objects

Cannot detect at compile time resulting in runtime error when retrieving

Explicit cast needed

A Parameterized Library class

The generic library class has a single type parameter E, allowing it to store objects of type E.

Using parameterized type E

Uses the services of parameterized ArrayList reference

Constructor

Allows any object of type E to be added

Objects retrieved are of type E

```
import java.util.*;

class Library<E>
{
   private List<E> resources = new ArrayList<E>();

   public void add(E x)
   {
      resources.add(x);
   }

   public E retrieveLast() {
      int size = resources.size();
      if (size > 0) {
         return resources.get(size-1);
      }
      return null;
   }
}
```

Using the Parameterized Library

- When using the parameterized (generic) Library class a type must be passes to the type parameter E.
- Note the element extracted from the parameterized Library need not be cast.

```
public class TestLibrary {
   public static void main(String args[]) {
      Library<Book> myBooks =  new Library<Book>();
      myBooks.add(new Book());
      myBooks.add(new Book());
      Book lastBook = myBooks.retrieveLast();
      lastBook.print();

      Library<Video> myVideos =  new Library<Video>();
      myVideos.add(new Video());
      myVideos.add(new Video());
      myVideos.add(new Video());
      Video lastVideo = myVideos.retrieveLast();
      lastVideo.print();
   }
}
```

Creating a Library to store Book objects

Creating a Library to store Video objects

No casting needed

Things to note when creating and using a Parameterized classes

- Primitive types cannot be passed as parameters.

```
List<Integer> numbers = new ArrayList<Integer>();
List<int> numbers = new ArrayList<int>();
```

- When a class uses parameterized type T, the type parameter T can be used as a reference but not for constructing.

```
T object = …
T[] a = …
         = new T();
         = new T[10];
```

- Though the parameter cannot be used as a constructor it can be used for casting

```
E e2=(E) new Object();
E[] e3 = (E[])new Object[10];
```

- Generic classes cannot be array base type (but can be a parameterized collection)

```
Library<Video> videoLibs[] =  new Library<Video>[10];
List<Video> vidLibs[] =  new ArrayList<Video>[10];
```

See http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf for more info.

A parameterized Stack class (without using other parameterized classes)

- A stack class that allows any objects of the same type to be stacked.
- It will initially create an array with initial capacity to store 10 elements – the capacity will be increased when necessary.
- Internally it creates an array of Object references and casts it to the parametric type as in:

```
private T[] elements;
…
public GStack(int capacity) {
    elements = (T[])new Object[capacity];
 }
```

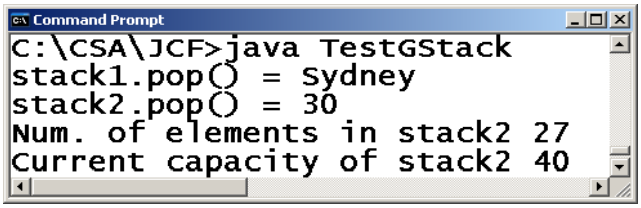
- Whenever capacity is increased it creates a new block of memory and copies the existing elements.


```
/* Adapted from Supplement Q: Generic Types By Y. Daniel Liang */
class GStack<T> {
   private T[] elements;
   private int size;

   // constructs a stack with the default capacity 10
   public GStack() { this(10); }

   // constructs a stack with the specified initial capacity
   public GStack(int capacity) {
      elements = (T[])new Object[capacity];
   }

   // puts the new element into the top of stack
   public T push(T value) {
      if (size >= elements.length)
         setCapacity(elements.length * 2);
      return elements[size++] = value;
   }

   // returns and removes the top element from the stack
   public T pop() {
      return elements[--size];
   }
   // returns top element without removing
   public T peek() {
      return elements[size - 1];
   }
```

```
   // tests whether the stack is empty
   public boolean empty() {
      return size == 0;
   }

   // returns the number of elements in the stack
   public int getSize() {
      return size;
   }

   // returns the capacity
   public int getCapacity() {
      return elements.length;
   }

   // sets new capacity - must be greater than current capacity
   public boolean setCapacity(int newCapacity) {
      if ( newCapacity > elements.length ) {
         T[] temp = (T[])new Object[elements.length * 2];
         System.arraycopy(elements, 0, temp, 0, elements.length);
         elements = temp;
         return true;
      }
      else
         return false;
   }
}
```


Using the generic Stack

```
public class TestGStack
{  public static void main(String args[])
   {  GStack<String> stack1 = new GStack<String>();
      stack1.push("Perth");
      stack1.push("Melbourne");
      stack1.push("Sydney");
      System.out.println("stack1.pop() = " + stack1.pop());

      GStack<Integer> stack2 = new GStack<Integer>();
      stack2.push(10);
      stack2.push(20);
      stack2.push(30);
      System.out.println("stack2.pop() = " + stack2.pop());
      for (int i=0; i<25; i++)
         stack2.push(i);
      System.out.println("Num. of elements in stack2 " +
                                              stack2.getSize());
      System.out.println("Current capacity of stack2 " +
                                              stack2.getCapacity());
   }
}
```



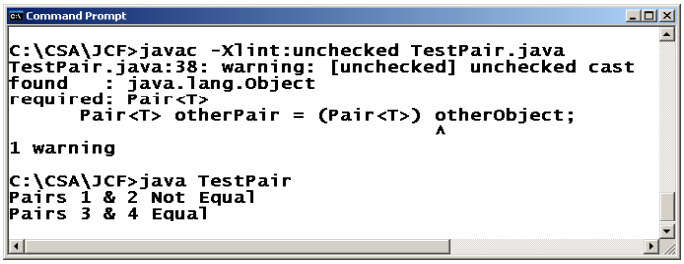
Comparison of Parametric classes

- The next program shows how a generic class could provide a method to compare two objects for equality. The Pair class keeps a pair of objects of the same type. It takes that type as a parameter.
- The equals method takes an Object reference to the other object. The Object reference is cast to the type of the current object using:
  - `Pair<T> otherPair = (Pair<T>) otherObject;`
- A cast such as this generates compiler warnings. To see the details of warnings compile with
  - `javac –Xlint:unchecked TestPair.java`
- The equals method first verifies it is an objects of the same class before comparing the parts that make up the two objects.
- See source code for better example of equals()


```
class Pair<T>
{
   private T first;
   private T second;
   public Pair()
   {
      first = null;
      second = null;
   }
   public Pair(T first, T second)
   {
      this.first = first;
      this.second = second;
   }
   public boolean equals(Object otherObject)
   {
      if (otherObject == null) return false;
      if (getClass() != otherObject.getClass())
         return false;
      Pair<T> otherPair = (Pair<T>) otherObject;
      return (first.equals(otherPair.first)
                && second.equals(otherPair.second));
   }
}
```

Testing the Pair class

```
public class TestPair
{  public static void main(String args[])
   {
      Pair<String> pair1 = new Pair<String>("10+5", "20+5");
      Pair<String> pair2 = new Pair<String>("15","25");
      if (pair1.equals(pair2))
         System.out.println("Pars 1 & 2 Equal");
      else System.out.println("Pairs 1 & 2 Not Equal");
      Pair<Integer> pair3 = new Pair<Integer>(10+5, 20+5);
      Pair<Integer> pair4 = new Pair<Integer>(15,25);
      if (pair3.equals(pair4))
         System.out.println("Pairs 3 & 4 Equal");
      else System.out.println("Pairs 3 & 4 Not Equal");
   }
}
```



Bounds for type parameters

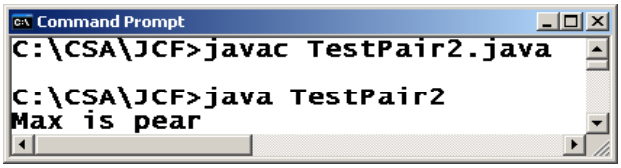
- Parametric classes may not make sense for all possible types
- In the parameterized library class we may want restrict the parameter type to Media or its subclasses.
- We may expect all items in the library to have a catalogue number or a method to get the expiry date.
- We can specify bounds for the type of parameters by using extends or implements clause (or both) as in:
  - `class Library<E extends Media> { …`
- As another example we extend the Pair class by incorporating a method named max() to return the bigger of its two objects.
- Java provides the Comparable<T> interface with a method public int compareTo(T other) that returns zero, negative or positive value depending on which object is larger.
- The Pair objects can be compared using this method if we specify bounds for the Parametric Pair class.
  - `class Pair<T extends Comparable<T>>`

Pair class with extends bound

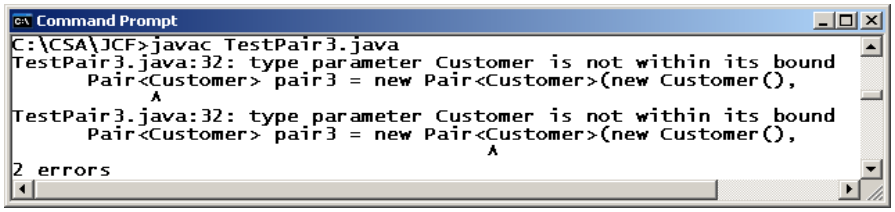
```
class Pair2<T extends Comparable<T>> {
   private T first;
   private T second;

   public Pair() {
      first = null;
      second = null;
   }
   public Pair(T first, T second) {
      this.first = first;
      this.second = second;
   }
   public T max() {
      if (first.compareTo(second) >= 0)
         return first;
      else
         return second;
   }
}
```


Testing the Bounded Pair class

```
public class TestPair2 {
  public static void main(String args[]) {
     Pair2<String> pair1 = new Pair2<String>("apple", "pear");
     System.out.println("Max is " +  pair1.max());
   }
}

class Customer {}

public class TestPair3 {
  public static void main(String args[]) {
      Pair2<Customer> pair3 = new Pair2<Customer>(new Customer(),
                              new Customer());
      System.out.println("Max is " +  pair3.max());
   }
}
```



Customer does not implement Comparable



A class with multiple type parameters

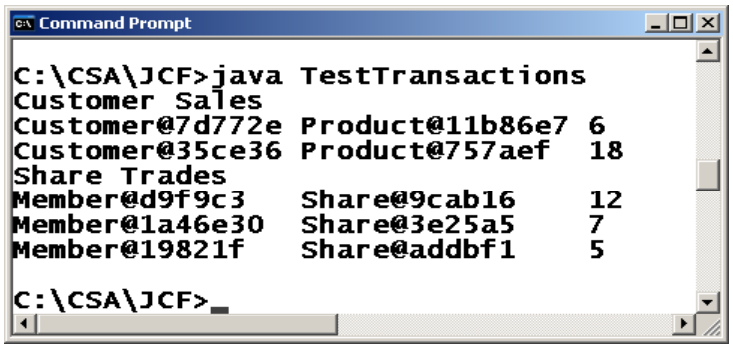
- As an example for a class with multiple type parameters we have created a Transaction class which associate one type of object with another type (Customer and Product or Member and Share) together with another to represent the quantity.
- Internally it maintains 3 arrays and a String to store the title to used in printing. The type of first two arrays depends on the type parameters passed. The third one is an Integer array to keep the quantity of transactions (share trades, sales …)

```
class Transactions<T1, T2> {
   private List<T1> owners = new ArrayList<T1>();
   private List<T2> items = new ArrayList<T2>();
   private List<Integer> nums= new
      ArrayList<Integer>();
   private String title;
   public Transactions(String title) { this.title = title; }
}
```



```
import java.util.*;
class Transactions<T1, T2>
{
   private List<T1> owners = new ArrayList<T1>();
   private List<T2> items = new ArrayList<T2>();
   private List<Integer> nums= new ArrayList<Integer>();
   private String title;
   public Transactions(String title) { this.title = title; }

   public void add(T1 owner, T2 item, int num)
   {
      owners.add(owner);
      items.add(item);
      nums.add(num);
   }
   public void list()
   {
      System.out.println(title);
      for (int i=0; i<owners.size(); i++)
        System.out.println(owners.get(i)+"\t"
           +items.get(i)+"\t"+nums.get(i));
   }
}
```

```
class Customer2 { }
class Product { }
class Member { }
class Share { }
public class TestTransactions {
  public static void main(String args[]) {
      Transactions<Customer2,Product> sales
            = new Transactions<Customer2,Product>("Customer Sales");
      Transactions<Member,Share> trades
            = new Transactions<Member,Share>("Share Trades");
      sales.add(new Customer2(), new Product(), 6);
      sales.add(new Customer2(), new Product(), 18);
      trades.add(new Member(), new Share(), 12);
      trades.add(new Member(), new Share(), 7);
      trades.add(new Member(), new Share(), 5);
      sales.list();
      trades.list();
   }
}
```



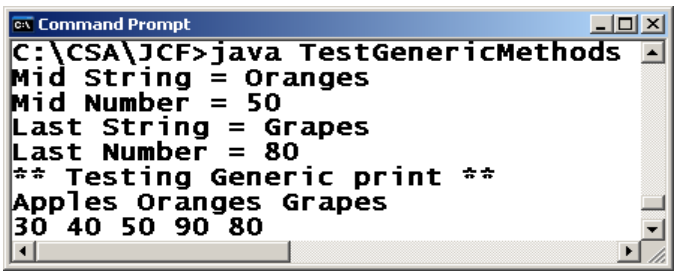
Generic Methods

- Generic methods can be member of generic classes or normal (ordinary) classes
- In the next example we have created a utility class that has three generic methods all declared as static.
- The method getMid() takes an array of elements of any type and returns the middle one.
- The method getLast() returns the last one.
- The print() method takes an array of elements of any type and prints them in a row.

```
class GenericMethods
{
   public static <T> T getMid(T[] a)
   {
      return a[a.length/2];
   }
   public static <T> T getLast(T[] a)
   {
      return a[a.length-1];
   }
   public static <T> void print(T[] a)
   {
      for (int i=0; i<a.length; i++)
         System.out.print(a[i] + " ");
      System.out.println();
   }
}
```



```
public class TestGenericMethods {
   public static void main(String args[]) {
      String s[] = { "Apples", "Oranges", "Grapes" };
      Integer nums[] = { 30 , 40 , 50, 90 , 80};

      String midS = GenMethods.<String>getMid(s);
      Integer midN = GenMethods.<Integer>getMid(nums);
      String lastS = GenMethods.<String>getLast(s);
      Integer lastNum = GenMethods.<Integer>getLast(nums);

      System.out.println("Mid String = " + midS);
      System.out.println("Mid Number = " + midN);
      System.out.println("Last String = " + lastS);
      System.out.println("Last Number = " + lastNum);
      System.out.println("** Testing Generic print **");
      GenMethods.<String>print(s);
      GenMethods.<Integer>print(nums);
   }
}
```



RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 5: Graphical User Interfaces I
Introduction, Event Handling

Lecture Slides

Graphical User Interfaces - Introduction

Most (modern) application programs (applications or apps) do not communicate with their users solely via text input and output. They use a **Graphical User Interface (GUI)**.

Most users find a **graphical user interface more convenient and easier to use** than a **text-mode console** system (recognition versus recall).

A **Graphic User Interface** (GUI, pronounced 'gooey') allows interaction via windows containing buttons, menus, control panels, dialog boxes, etc.

However, it is important to note that a **good user interface cannot compensate for a poor program.**

Likewise, an otherwise excellent program **can be rendered unusable or useless** by a poor user interface (see following slide, usability).

Therefore, it is important to design an **effective means** for the **user to interact** with the program.

Designing for Usability

Usability is a combination of the following user-oriented characteristics:

(Designing the User Interface 3rd Ed., Shneiderman, B., 1998)

- High speed of user task performance.
- Low user error rate.
- Ease of learning.
- User retention over time.
- Subjective user satisfaction.

The user is the final judge of the usability and appropriateness of the interface.

Achieving usability requires attention to two main components:

- The product (interactive software system)
- The process by which the product is developed.

Is primarily the domain of usability specialists and graphic designers. This course emphasis quality of the user interface ***code***

Java's GUI

Java's Abstract Window Toolkit (AWT) and Swing are standard Java libraries of packages and classes designed to provide a framework for programmers to create GUIs. AWT provides the basic GUI classes. Swing extends AWT and provides more advanced GUI components. They share the same concepts and event model. More on Swing in the next slide.

Where available Swing is preferred over AWT however AWT is still used for 2D graphics, fonts, color, layout management and event handling (since Swing does not replace or supersede these AWT features).

This course:

- Covers general software design principles
- Introduces AWT (especially the core event handling and layout management model)
- Introduces basic Swing components for designing the graphical user interface

NOTE: IBM provides an alternative GUI framework called SWT (Standard Widget Toolkit) which is not covered in this course (Eclipse is a good example of an SWT project).

Swing Basics

Swing is an additional graphical interface toolkit (distributed as part of Java SE since Java 1.2) that provides many improvements over AWT that was introduced with Java 1.0 and refined with Java 1.1.

Swing is built on top of the AWT component set, and relies on it for its basic functionality.

Swing and AWT components, however, should not be used together in the same interface (due to z-order incompatibility).

The components and other classes related to AWT are in the series of packages named *java.awt.**

The components and other classes related to Swing are in the series of packages named *javax.swing.**

Swing UI components begin with 'J' to differentiate them from their AWT counterparts; e.g. the Swing button object is JButton, the AWT equivalent is Button.

Lightweight and Heavyweight Components

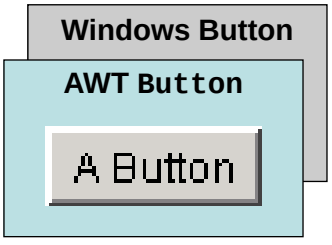
Most Swing components are "lightweight" components, compared to the AWT (where they are "heavyweight".)

Lightweight components are those that are written completely in Java, and do not rely at all on the host operating system.

Heavyweight components depend on the related 'peer' component of the native operating system.

This is why AWT-based programs look slightly different on each platform, but Swing programs maintain the same look (and also why these component sets should not be mixed together.)

The GUIs we develop in this course will be Swing based.



Heavyweights rely on their native peers.

Applets and Applications

There are **two kinds** of program in Java SE: **applets** and **applications.**

Applets are **container-based** applications which usually **run in a web browser.** An applet is typically downloaded from the network (or internet) by a web browser to execute locally on your computer. There are a number of (mainly security related) **restrictions that are placed on applets** (discussed later). They are always GUI based.

Java applications can be console (text) or GUI based**. GUI applications** are **standard Java programs** which create a graphical user interface. These can be run from the command prompt and are sometimes called **stand-alone** applications.

The process of creating a graphical user interface for applets and applications is very similar and both use the same AWT and Swing components.

JApplets and Basic Graphics

JApplets (Swing applets) are created by extending the JApplet class:

```
public class MyApplet extends JApplet
```

There are a number of methods (see slide 12 for details) inherited from JApplet which can be overridden to provide custom behaviour e.g.:

init() – this is the place for initialising the applet.
paint() – this controls the (re-)drawing of the applet.

The paint() or paintComponent() method can be used simply to make the applet a drawing canvas, using the predefined Java drawing methods of the Graphics class. However more often AWT or Swing components are used to create the user interface (especially for business type applications cf. games).

Note that JApplet extends java.applet.Applet (AWT Applet) and thus much of its functionality comes from AWT.

Swing Applets (1)

Let's compare a simple text I/O application with a GUI applet. Recall the famous HelloWorld program below.

```
public class HelloWorld {

  public static void main(String[] args) {

    System.out.println("Hello world");
  }
}
```

Calling method (function) ***println*** on the attribute ***out*** of the class ***System***.

NB. ***out*** is an ***OutputStream*** object. ***System.out*** represents the standard output (console).

Let's then do the same as a GUI …

Swing Applets (2)

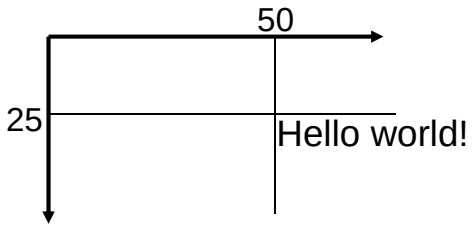
```
import javax.swing.JApplet;
import java.awt.Graphics;

public class HelloWorld extends JApplet {

  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

Import from swing and awt packages

Extend JApplet class

"Automatic" container based method; no main() method in an applet.

Window coordinates in pixel units – (0,0)



Running Applets

Save applet code in a file, eg. HelloWorld.java

Compile to produce HelloWorld.class

Save below in a file: HelloWorld.html

```
<HTML>
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET></HTML>
```

Open a browser and load HelloWorld.html.

This will run the applet. (Browser must be enabled to run Java)

Alternatively, open ***appletviewer*** that comes with Sun's free JDK.

The compiled .class file is downloaded (from across the Internet if necessary, depending on the path given in Applet Code tag) and interpreted in the local computer by the browser.

Applets – Swing and AWT

Older browsers may not support Swing components. In this case you need to install the Java plug-in (see http://java.sun.com/products/plugin/)

As far as this course is concerned, it is sufficient to test your applets entirely in *appletviewer*

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {

  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

In order to write Hello World as an AWT applet, some minor changes are necessary.

The Applet Life Cycle

Method Name	Executed by browser when an HTML document that contains the applet...	Is often used to...
init	Is loaded by the browser	Set up user-interface, start conditions
start	Is displayed by the browser or revisited and displayed	Re-initialise suspended functionalities
stop	Is closed by the browser or the browser is shut down	Suspend (resource-consuming) functionalities
destroy	After stop	Free up resources

Method Name	Executed by browser when the applet need redrawn, the named method...	Is often used for...
repaint	Redraws the applet. Calls method *update*	Advanced drawing and animation techniques
update	Clears the applet's graphics area. Calls method *paint*	Advanced drawing and animation techniques
paint	Redraws the applet graphics area	Simple drawings

Another Example: Bouncing Ball Applet (1)

```
import javax.swing.JApplet;
import java.awt.*;

/* <APPLET CODE="Ball.class" HEIGHT=300 WIDTH=300>
   </APPLET> */

public class Ball extends JApplet {
  private int x = 0, y = 0, dx = 1, dy = 2;
  private int diameter = 50, width = 0, height = 0;

  public void init() {
    width = getSize().width;
    height = getSize().height;
  }
```

The commented-out HTML code is a short cut to run an applet through appletviewer. No need for an html file. Simply type:

% appletviewer Ball.java

This code gets the size of the applet surface.

Another Example: Bouncing Ball Applet (2)

```
  public void paint(Graphics g){
    while (true) {
      for (int i=0; i<10000; i++);
      x += dx;
      if (x < 0) dx = 1;
      if (x > width-diameter) dx = -1;
      y += dy;
      if (y < 0) dy = 2;
      if (y > height-diameter) dy = -2;
      g.setColor(this.getBackground());
      g.fillRect(0,0,width,height);
      g.setColor(Color.red);
      g.fillOval(x,y,diameter,diameter);
    }
  }
}
```

this loop will provide a short delay.

this code will clear the graphics area.

Applet Security

For security purposes, under normal circumstances, applets run in a sandbox which provides them limited access to the machine;

for example, an applet cannot read or write the local file system or communicate with hosts other than that from which it was downloaded.

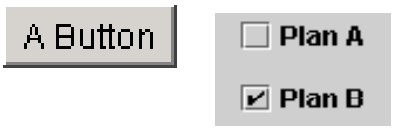
In special cases where these security provisions need to be overruled, the applet can be run with options which specify the actions to be allowed.

Often this is accompanied by a signed certificate to verify the applet provider (e.g. VeriSign certificates) however this is beyond the scope of this course.

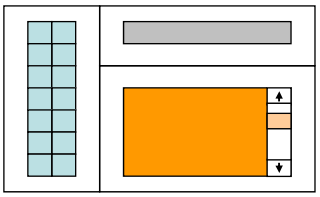
Developing a GUI

There are **three basic things** that are necessary to develop a graphical user interface:

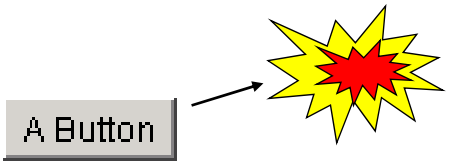
The GUI components: these are the buttons, labels, check boxes, etc. which form the interface.



Component arrangement: this is the scheme whereby the UI components are arranged to create the interface.



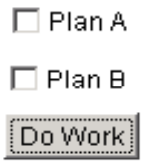
Response to user requests: this is the act of associating actions to user requests (known as 'events'.)



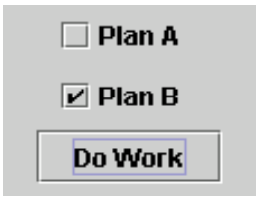
AWT and Swing Components

There are two sets of components in Java: the **AWT** (Abstract Windowing Toolkit), and the **Swing component set**.

The **AWT component set** is a very **basic set** of user interface items.



The **Swing component set** (which is an extension of the AWT set) are **far richer** in presentation and functionality, however they are **only available in later Java versions.** Most web browsers **cannot use Swing** without plug-ins; many applets use AWT.



Mixing AWT and Swing components **is not recommended!**

Components and Containers

The AWT set is structured in a class hierarchy; **every UI component** is a **descendant of the `Component` class**. This contains the **very basic functionality** common to all components, such as the **ability to be laid out** on an interface, **foreground and background colours**, etc.

All Swing components (both atomic and non-atomic) are derived from the class **java.awt.Container**.

All Swing components whose names begin with "**J**", except for the top-level containers, are derived from **javax.swing.JComponent**

A **Container** is a special type of component to which you can add other components. Example, JFrame, JApplet, JPanel etc. All containers have methods to add and remove components

```
public Component add (Component comp);
public void remove (Component comp);
```

as well methods to arrange these components (more on this next lecture)

The Swing Containment Hierarchy

A Swing containment hierarchy typically has at least:

a top-level container: the root container that holds everything together. e.g. JApplet or JFrame

one or more intermediate containers to simplify the positioning of atomic components. e.g. JPanel, JScrollPane

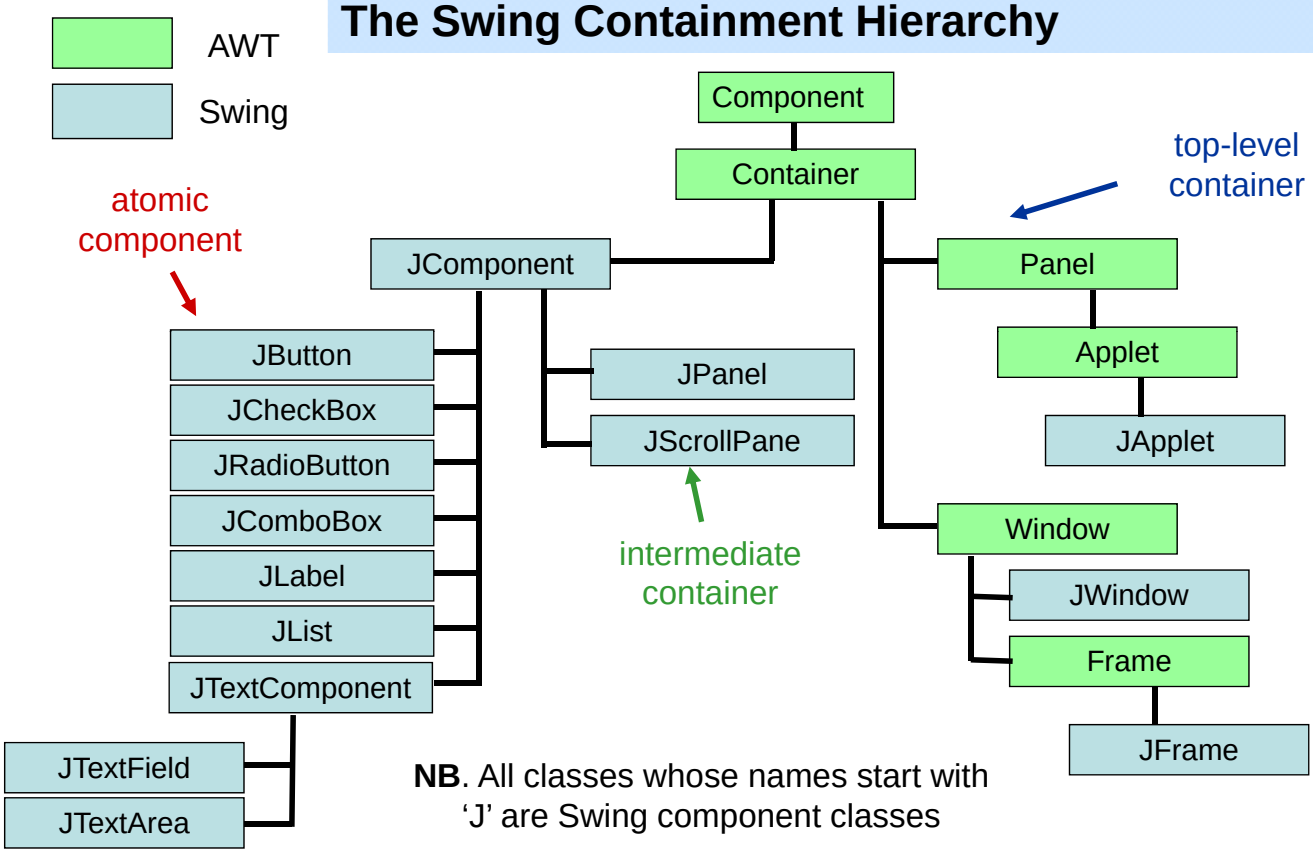
an atomic component: a self-sufficient component not holding other components. eg. JButton, JLabel

In a Swing application:

The top-level container must be Swing. e.g, JApplet (not Applet)

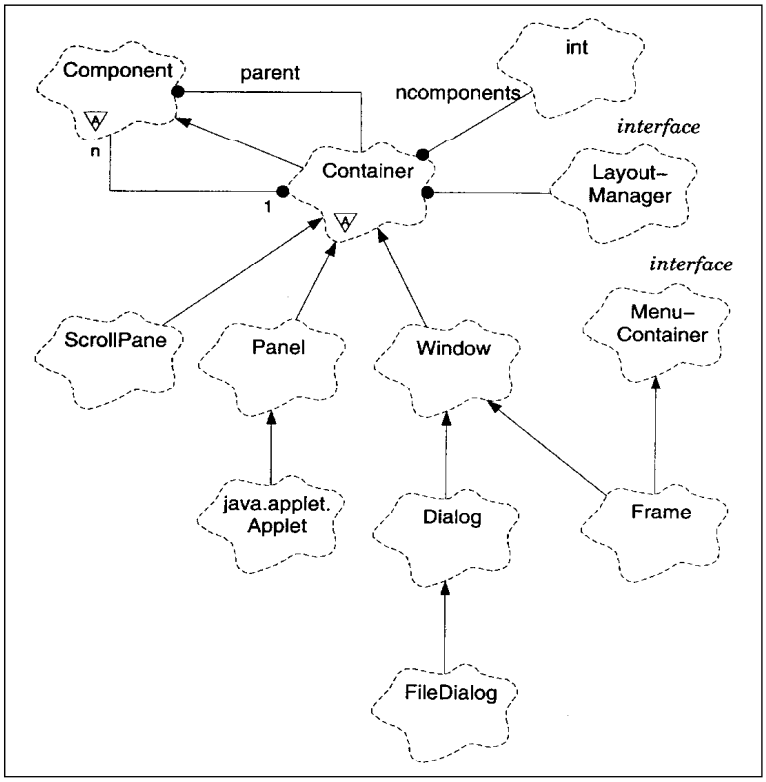
A component cannot be added directly to the top-level container, but must be added to an intermediate container contained in the top-level container, called the content pane. This is done by calling the method public Container. getContentPane() that returns a reference to its content pane (which is actually a JPanel).

The Swing Containment Hierarchy



NB. All classes whose names start with 'J' are Swing component classes

Class Diagram – Component / Container Relationships



Hello World, Revisited

The simple “Hello world!” program, implemented using a **`JLabel`** GUI component for the text:

```
import javax.swing.*;

public class HelloWorld extends JFrame {

  JLabel greeting = new JLabel("Hello world!");

  public HelloWorld() {
    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    c.add(greeting);
    setVisible(true);
  }
  public static void main(String args[])
  {
    new HelloWorld();
  }
}
```

Top-level Container

Atomic Component

Intermediate Container

GUI Component Example (1)

```
import java.awt.*;
import javax.swing.*;
public class GUIComponents extends JFrame {

  JPanel pn;
  JButton button = new JButton("Button");
  JCheckBox checkBox = new JCheckBox();
  JLabel label = new JLabel("Label");
  JTextField textField = new JTextField("Text Field");
  JTextArea textArea = new JTextArea("Text Area", 5, 20);

  public void GUIComponents() {
    super();
    Container container = getContentPane();
    container.setLayout(new FlowLayout());
    container.add(button);
    container.add(checkBox));
    container.add(label);
```

GUI Component Example (2)

```
pn = new JPanel();
pn.setBackground(Color.red);
container.add(pn);
container.add(textField);
container.add(textArea);
DefaultListModel data = new DefaultListModel();
data.addElement("item 1"); data.addElement("item 2");
data.addElement("item 3"); data.addElement("item 4");
data.addElement("item 5"); data.addElement("item 6");
data.addElement("item 7"); data.addElement("item 8");
data.addElement("item 9"); data.addElement("item 10");
JList list = new JList(data);
JScrollPane scroll = new JScrollPane(list);
container.add(scroll);
JComboBox selector = new JComboBox();
selector.addItem("Yes");
selector.addItem("No");
selector.addItem("Maybe");
container.add(selector);
```

This code sets up the list.

This code sets up a drop-down box.

GUI Component Example (3)

```
JRadioButton oft = new JRadioButton("often", true);
JRadioButton st = new JRadioButton("sometimes");
JRadioButton never = new JRadioButton("never");
ButtonGroup group = new ButtonGroup();
group.add(oft);
group.add(st);
group.add(never);

container.add(oft);
container.add(st);
container.add(never);
  }
}
```

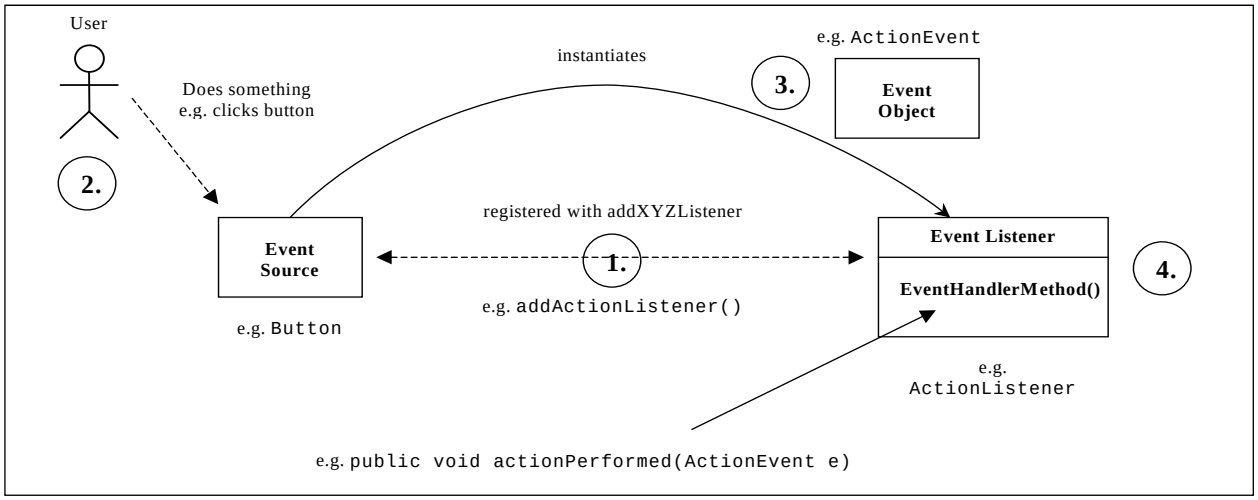
A ButtonGroup ensures only one item in the group can be selected.

AWT - Sources, Listeners and Events

- Delegation event model in AWT 1.1 (and later)
  - Event sources (usually `Components`) fire *events*
  - Events are listened for and acted upon by *event listeners*
  - Listeners are registered with a source by invoking the component's `addXYZlistener(XYZlistener)` method
  - Can add multiple listeners to an event source
    - Notification order is undefined
  - Can register single listener with multiple sources (careful you don't reduce cohesion!)
  - Event listeners implement *event handling methods*
    - are passed an instance of an *event object* (an instance of `java.awt.Event`)
    - contains info. about the event and a reference to the event source

Diagram of Event Interaction

1. A listener is registered with an event source [e.g button.addActionListener(new ActionListener())]
2. The user interacts with the event source (e.g. clicks button with mouse)
3. The event source instantiates an event object containing information about the event
4. The event source passes this event object to the event listener (i.e. invokes the event handler method of its registered listener and passes it the event object as a parameter)



Components as Event Sources

- All Java components inherit the following listener registration methods (addXYZListener) from their parent class java.awt.Component:
  - `void addFocusListener(FocusListener)`
  - `void addKeyListener(KeyListener)`
  - `void addMouseListener(ComponentListener)`
  - `void addMouseMotionListener(MouseMotionListener)`
  - `void addInputMethodListener(InputMethodListener)`
  - `void addComponentListener(ComponentListener)`

- Each AWT component also has specialised listener registration methods:
  - e.g. MenuItem and Button have `addActionListener` methods
  - Usage: `Source.addXYZListener(XYZListener listener)`
  - see table on next slide for complete list

Listeners

- java.awt.event package defines many interfaces for different types of listeners.
- Each interface defines methods that are called when a specific event occurs e.g.
  - `ActionListener` defines a lone method:
    `void actionPerformed(ActionEvent)`
  - invoked when an action event occurs within the source (component) that registered the `ActionListener` with `Component.addActionListener(ActionListener l))`
- Instances of classes that are to act as event listeners must implement one or more of the listener interfaces

Event Objects

- `EventObject` is a simple class
  - base class for all of the `AWTEvent` classes
  - keeps track of its event source, i.e. the object that triggered the event
  - use `EventObject.getSource()` method to return the event source [must be converted from Object to correct type e.g Window `w=(Window)e.getSource()`]
- Declare constants to provide additional info about the event
  - `FOCUS_LOST, FOCUS_GAINED` etc.
  - use methods such as `AWTEvent.getID()` to return constants
- Provide methods to access event properties
  - e.g ActionEvent class
    - `String getActionCommand();`
    - `int getModifiers();`

Event Handling Example

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class ButtonTest extends Applet {
    public void init() {
        Button button = new Button("Press Me");
        button.addMouseListener(new
                    ButtonMouseListener());
        add(button);
    }
}
```

Event Handling Example (cont)

```
class ButtonMouseListener implements MouseListener {
    public void mouseEntered(MouseEvent event) {
        System.out.println("Mouse Entered Button");
    }
    public void mouseExited(MouseEvent event) {
        System.out.println("Mouse Exited Button");
    }
    public void mousePressed (MouseEvent event) { }
    public void mouseClicked (MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
}
```


AWT Adapters

- Most AWT Listener interfaces provide more than one method which must be implemented
- Can be tedious if only one or two are of interest
- e.g. interested only in `WindowListener.windowClosing()` but not the other six Window Events
- Consequently AWT provides Adapters for each of the Listener interfaces that provide multiple methods
- Adapters provide no-op (empty) implementations of the listeners
- => Specific methods can be overridden and handled and the others ignored
- Adapters are classes not interfaces (i.e. they are extended not implemented) therefore only one adapter can be used per class

Adapter Example

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class ButtonTest2 extends Applet {
  public void init() {
      Button button = new Button("Press Me");
      button.addMouseListener(new
             ButtonMouseListener());
      add(button);
  }
}
```


Adapter Example (contd.)

```
class ButtonMouseListener extends MouseAdapter {
  public void mouseEntered(MouseEvent event) {
      System.out.println("Mouse Entered Button");
  }
  public void mouseExited(MouseEvent event) {
      System.out.println("Mouse Exited Button");
  }
}
```

Focus Events

- Maximum of one component can have focus at any one time
- Keyboard events are sent to the focused component
- Typically have a prominent appearance
- Qualified as `FOCUS_LOST` or `FOCUS_GAINED` events
- Focus event can be permanent or temporary
  - permanent focus event is when focus is deliberately changed
  - temporary event occurs when containing object (usually window) loses focus
  - `FocusEvent.isTemporary()`
- Gain focus by:
  - interacting with the component
  - invoking `Component.requestFocus()`
  - typing TAB or SHIFT-TAB
- Use `FocusEvent` and `FocusListener` to handle the focus

Key Events

- Key Events fired when a key is pressed or released in a component that has the focus
- KeyListener interface has KeyPressed(), KeyReleased() and keyTyped() methods
- Each key on the keyboard has a unique key code
  - Returned by `KeyEvent.getKeyCode()` (e.g. `KeyEvent.VK_A`, `KeyEvent.VK_F1` etc. - see API docs for full list of key codes)
  - For key typed events, keyCode is `KeyEvent.VK_UNDEFINED`
- Keys also map to a UNICODE (16 bit) character
  - `KeyEvent.getKeyChar()`
  - Returns the character associated with the key in this event. For example, the key-typed event for shift + "a" returns the value for "A"
  - If no valid Unicode character exists for the key then it returns `KeyEvent.CHAR_UNDEFINED`

Mouse and Mouse Motion Events

- Mouse events and mouse motion events are distinguished via separate `MouseListener` and `MouseMotionListener` interfaces
- Mouse moved and mouse dragged are mouse motion events
- The remaining events (enter/exit, pressed/released and clicked) are mouse events
- Both share the same event class MouseEvent
  - can identify position of cursor: `getPoint()` or `getX(), getY()`
  - can determine key modifiers if any (SHIFT, CTRL etc.) `getModifiers()`
  - can determine click count (can be more than two!) `getClickCount()`

Window Events

- Fired by instances of `java.awt.Window` (and subclasses such as `Frame` and `Dialog`)
- Signify that a window has been *activated/deactivated, iconified/deiconified, opened/closed*, or closing
- `windowClosing()` is commonly overridden for a main frame window to exit the associated application when the frame is closed`

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

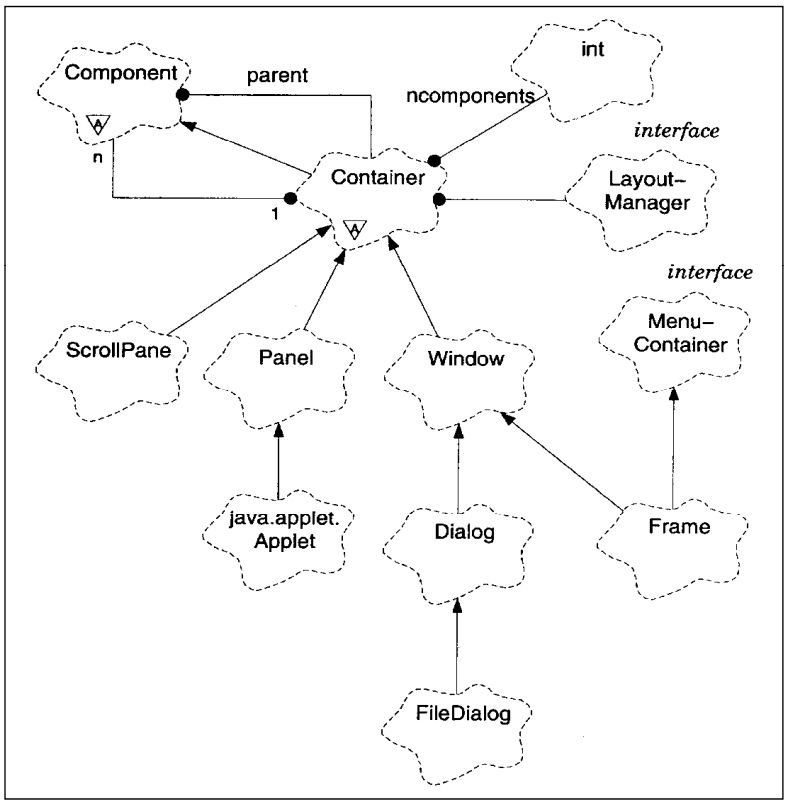
Topic 6: User Interfaces and Graphics II
Layout Management, Windows and Menus

Lecture Slides

Containers, Components & Layout Management

- The relationship between containers, their contained components and their layout manager is fundamental to the AWT (and Swing)
- A container is an AWT component that can contain other components (added using the `Container.add(Component)` method)
- The abstract class `java.awt.Container` extends `Component` and thus Containers may also contain other Containers (composite pattern)
- Containers group related components and treat them as a unit, which helps arrange components on the display
- Many AWT components and all Swing components subclass container (e.g. (J)Frame, (J)Applet, (J)Panel etc.)

Class Diagram – Component / Container Relationships



Layout Managers

- Containers are not responsible for laying out (sizing, positioning etc.) their own components => all containers have an instance of a layout manager
- A Layout Manager is a class that implements the `java.awt.LayoutManager` or `java.awt.LayoutManager2` interface
  - the LayoutManager2 interface allows a component to provide constraints that assist with resizing
  - uses overloaded `Container.add(Component comp, Object constraints)` method
- AWT provides 5 pre-defined layout managers
  - `FlowLayout*, BorderLayout*, GridLayout*, CardLayout` and `GridbagLayout` (* covered in this lecture)
  - Swing adds the `BoxLayout` and `SpringLayout` (low level for use with GUI Builders)
- Call Container's `void setLayout(LayoutManager mgr)` method to set a containers layout manager

Why use Layout Managers?

- Layout managers are used in Java in preference to fixed x,y,width,height positioning:
  - fixed positioning can be achieved by setting the layout manager to ‘null’
  - poor solution for windows that need to be resized because of font metrics, placement etc. => must be manually calculated (difficult!)
  - cross platform programming is tedious (because of differences in available fonts, component sizes etc.)
  - layout managers provide solutions for common layout scenarios (i.e. reuse instead of coding from scratch)

Component Preferred/Minimum Sizes

- components can specify *preferred* and *minimum* sizes that are used by the layout manager
- *default values automatically set by the component (or peer)
- can be changed in AWT but requires subclassing in order to override two Container methods:
  - `public Dimension getPreferredSize()`
  - `public Dimension getMinimumSize()`
  - Swing provides setter methods
- not all layout managers always abide by the components preferred and minimum sizes
- preferred and minimum sizes are usually set by the peer so generally don’t need to be set manually
- there is also a *maximum* size but is not used by standard AWT layout managers

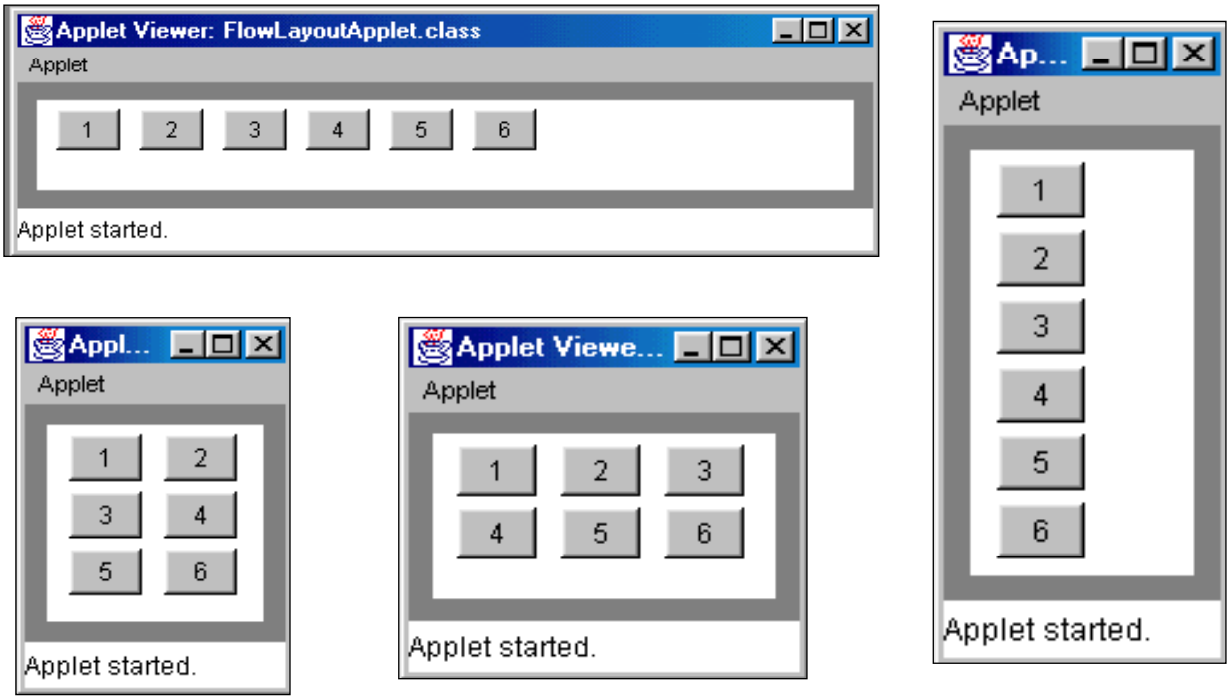
Flow Layout

- displays components left to right, top to bottom.
- default layout manager for panels and applets
- respects preferred size and height if component has not been explicitly sized with `setSize()`
- can set the spacing between components during construction or with `setHgap(int hgap)` or `setVgap(int vgap)`
- can set the alignment of components during construction or with `setAlignment(int align)`
  - `FlowLayout.CENTER, FlowLayout.LEFT, FlowLayout.RIGHT`

Contructors

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, int hgap, int vgap)`

Flow Layout Resizing



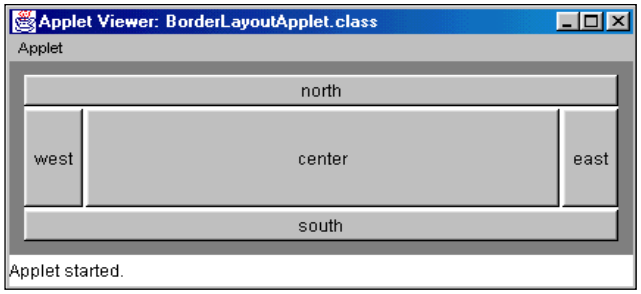
Border Layout

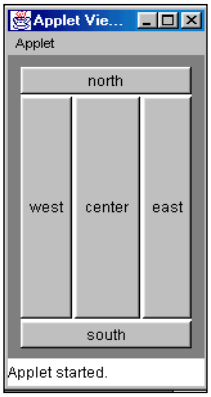
- Lays out at most five components into geometric regions
- Implements `LayoutManager2` interface so requires constraints which are string constants :
  - `BorderLayout.NORTH , BorderLayout.SOUTH , BorderLayout.WEST, BorderLayout.EAST & BorderLayout.CENTER`
- *north* and *south* components are stretched horizontally but preferred height is respected
- *west* and *east* components are stretched vertically but preferred width is respected
- *center* layout gets whatever space is left (preferred size is ignored)
- vertical and horizontal gap set in same manner as FlowLayout

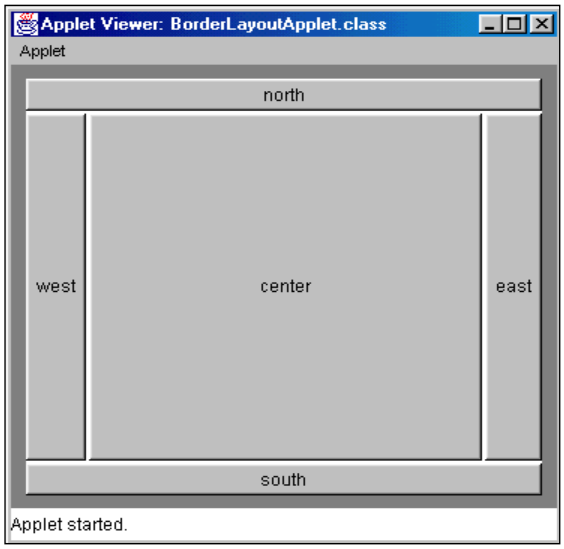
Constructors

- `BorderLayout()`
- `BorderLayout(int hgap, int vgap)`

Border Layout Resizing



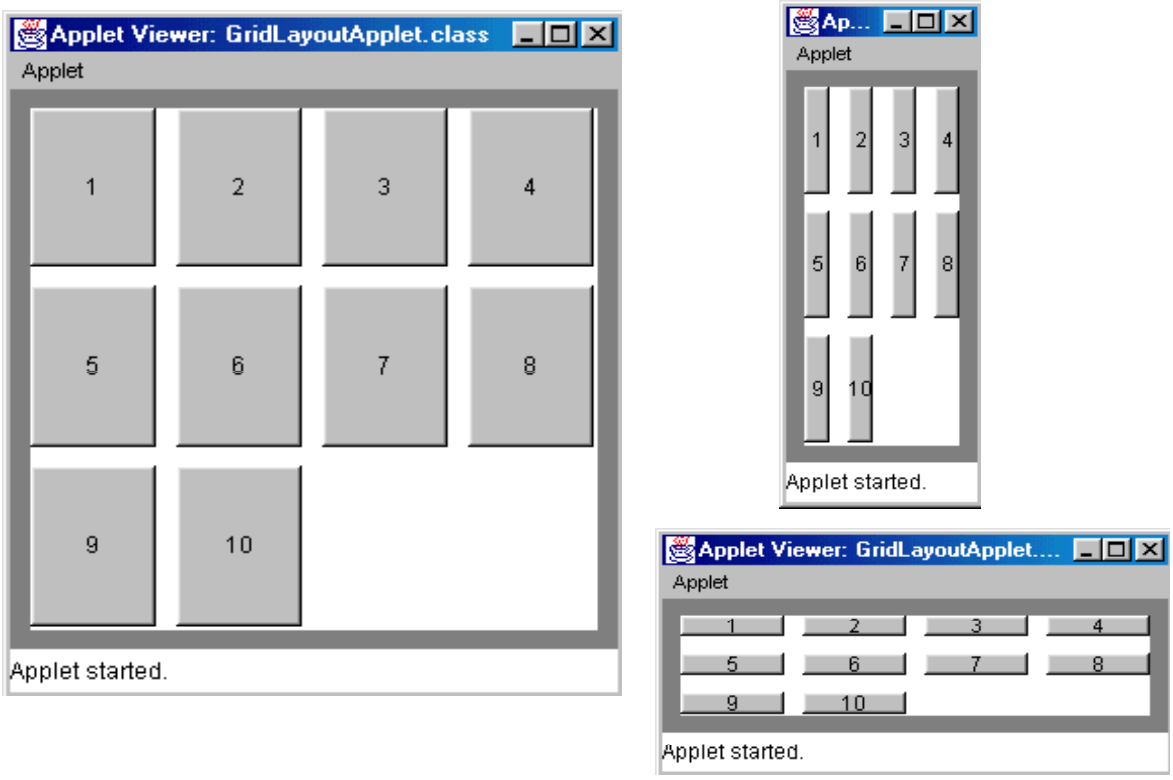




Grid Layout

- Lays out components in a grid
- Implements `LayoutManager` interface (i.e. there are no constraints)
- Number of cells are specified during construction
  - if a row or column is specified as 0 then size is automatically computed depending upon number of contained components
  - row and column cannot both be zero (throws exception)
- All cells are the same size (components grow or shrink to fill the display area)
  => preferred and *minimum* sizes of contained components are ignored
- Can set vertical gaps and horizontal gaps as with `BorderLayout` and `FlowLayout`
- Constructors:
  - `GridLayout()`
  - `GridLayout(int rows, int cols)`
  - `GridLayout(int rows, int cols, int hgap, int vgap)`

Grid Layout



GridBagLayout

- Most flexible but most complicated* of the layout managers
- Implements `LayoutManager2` interface
  - constraint is an instance of `GridbagConstraints` class
- Lays out components in a grid
  - components may span more than one cell
  - number of cells is not specified at construction but rather determined by the constraints which are set for each component
  - size of cells is not specified but is determined in relation to the size of the component it contains
- Particularly useful for laying out input forms
- Constructor
  - `GridBagLayout()`
- Not covered in Programming 2

Windows

- `java.awt.Window` provides a superclass with common functionality that is extended by both `Frame` and `Dialog`
- it subclasses `Container` allowing components to be added with the `add()` method
- generally not instantiated directly
- provides a number of general purpose window methods
  - inherited by `Frame` and `Dialog`
  - recall `WindowListener` and `WindowEvent` classes that are used with Windows

java.awt.Frame

- `java.awt.Frame` is a subclass of `java.awt.Window`
  - provides a border, title and optional menubar
  - may be resizable (depends on implementation)
  - can be minimized to an icon (depends on implementation)
  - can have an associated icon image: `setIconImage(Image image)`
- constructor: `Frame(String Title)`
  - *Title* represents the text to be displayed in the title bar
- initially invisible and remains so until `setVisible(true)` method is called
- Swing `JFrame` extends `java.awt.Frame` and has same basic behaviour
  - main difference is components are added to contentPane [retrieved via `getContentPane()`] not directly to `JFrame`
  - usually add a container (`JPanel`) to contentPane and then manage components in the `JPanel`

java.awt.Dialog

- `java.awt.Dialog`
  - similar to frame (provides a border and title and is resizable)
  - however it cannot have a menu or be iconized
  - must be anchored to a frame (same as Window)
- Dialogs can be modal => when `setVisble(true)` is called:
  - input to the Dialog's ancestors (usually a `Frame`) is blocked
  - the thread that launched the dialog is suspended until the dialog is closed
  - non-modal behaviour is default if not specified with constructor
- AWT provides limited Dialog support
  - `FileDialog` is the only custom dialog
  - message, yes/no and question dialogs must be hand coded
  - Swing provides JOptionPane class with static methods for creating basic dialogs
  - e.g. `JOptionPane.showMessageDialog(...)`
- Relationship between Swing `JDialog` and AWT `Dialog` is same as `Frame/JFrame`

Sizing and Moving Windows

- All windows (i.e. frames and dialogs) inherit from `Component` => use the methods of component class for sizing etc.

```
void setLocation(int x, int y)
void setSize(int width, int height)
void setBounds(int x, int y, int width, int height)
e.g.      Frame f=new Frame("Test");
          f.setSize(100,100);
          f.setLocation(0,0);
          f.show();
```

- Usually only size top level containers (Frame, Applet etc.) otherwise use layout management for sizing

Example: JMenuBarDemo (Pull down menus)

For additional help using menus see
http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html

Also look at **JPopupMenu** (which is not covered explicitly in this lecture)

In this example we will look at incorporating a menu bar into a GUI application.

New concepts covered in this example include:

- Adding a Swing JMenuBar to a JFrame
- Creating and manipulating JMenuItem components.
- Using a FileChooser to interact with the underlying file system
- Using a DialogBox to display output to the user
- Adding a JScrollPane to a container

We will also explore a different way of setting up event Listeners for components in your GUI – namely using (anonymous) inner classes.

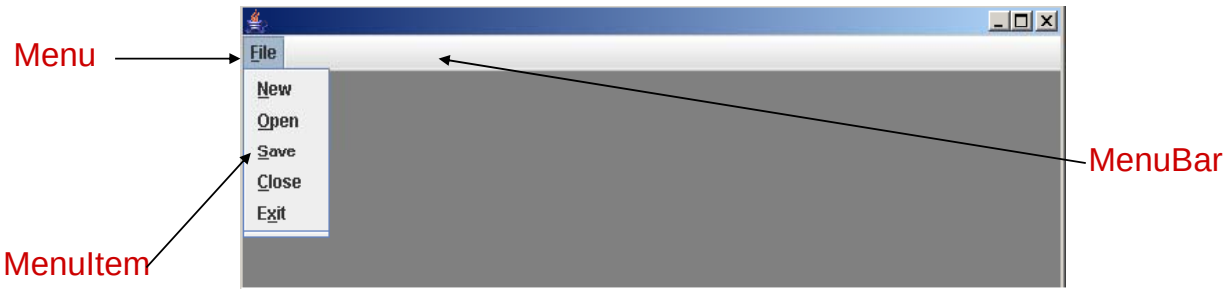
Adding a Menu Bar to a JFrame

A Menu bar consists of three different parts – the J/MenuBar component itself, one or more J/Menu components (which are added to the J/MenuBar) and a set of J/MenuItem components for each Menu component which represent the menu options.

To add a menu bar to a JFrame as shown in the example below you need to create the J/MenuBar, create one or more Menu components, fill each of the Menu components with their corresponding J/MenuItem components and add each of the (now filled) Menu components to the J/MenuBar.

Once the MenuBar has been set up it can be added to the JFrame using the method:

void setJMenuBar(JMenuBar menubar);



Example: Setting up a JMenuBar for a JFrame

To set up a menu bar you need to create the JMenuBar component, create one or more Menu components, create and add the required MenuItem components to the Menu Objects and then add each of the Menu objects to the JMenuBar, after which the JMenuBar can be added to your JFrame.

```
import java.awt.*; import java.awt.event.*; import java.io.*;
import javax.swing.*; import javax.swing.filechooser.*;

class JMenuBarDemo extends JFrame {
   private JTextArea text; private JScrollPane scroll;
   private JMenuBar menu;  private JMenu fileMenu;

   private JMenuItem newItem;  private JMenuItem openItem;
   private JMenuItem saveItem; private JMenuItem closeItem;
   private JMenuItem exitItem; private JMenuItem lastFileItem;

   private final JFileChooser fileSelection;
   private Container contentPane;
   private boolean fileOpen;
```

JMenuBarDemo Example: Setting up the menu components

```
public JMenuBarDemo()
{
   fileOpen = false; menu = new JMenuBar();
   fileMenu = new JMenu("File");

   fileMenu.setMnemonic(KeyEvent.VK_F);
   menu.add(fileMenu);

   newItem   = new JMenuItem("New", KeyEvent.VK_N);
   openItem  = new JMenuItem("Open", KeyEvent.VK_O);
   saveItem  = new JMenuItem("Save", KeyEvent.VK_S);
   closeItem = new JMenuItem("Close", KeyEvent.VK_C);
   exitItem  = new JMenuItem("Exit", KeyEvent.VK_X);


   fileSelection = new JFileChooser(new File("H:\\"));
   text = new JTextArea(20,50);
   text.setFont(new Font("Courier New",Font.PLAIN,12));

   scroll = new JScrollPane(text);
```

This is what is called a "mnemonic" - it is basically a shortcut key for the menu (this shortcut is "Alt-F").

Menu items can also have mnemonics

Create a file selection dialog for later use

Add a scroll pane to the TextArea component

JMenuBarDemo example: Adding a listener using an inner class

An (anonymous) inner class if effectively a one-use class that is directly associated with another class (or object). In an event-driven program it is a quick and easy way to add the event handling (listener) code directly to a specific component – thus there is no need to identify individual components in the listener code later on.

```
newItem.addActionListener(
              new ActionListener(){

public void actionPerformed(ActionEvent e) {

      if (!fileOpen) {
         fileOpen = true;
         contentPane.add(scroll);
         validate();
         contentPane.repaint();
      }
      else {
         text.setText("");
      }
   }});
```

Inner class definition

Listener for the "New" menu item

When we add components to a container at runtime we need to call the validate() method to make sure everything is laid out correctly

We also need to repaint the content pane to ensure that the background of the text area is "repainted"

JMenuBarDemo example - Using a JFileChooser / dialog

A file selection dialog is a popup window that you can use to explore the local fiile system and select files for use within your application. At the beginning of this example we created a new JFileChooser object as shown below – we can use this JFileChooser object as a selector for both "save" files and files we want to open.

```
fileSelection = new JFileChooser(new File("H:\\"));
```

We passed a new File object to the constructor above which represents the initial path the JFileChooser will start from. Now lets have a look at the code for the "open" listener.

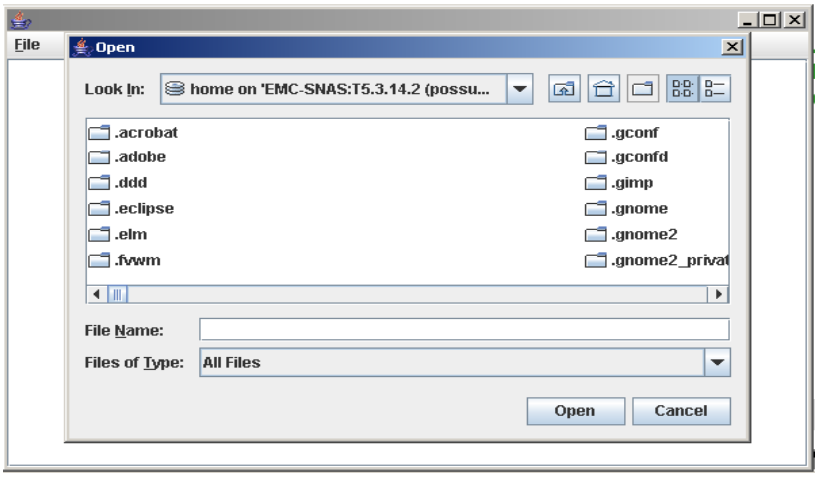
```
openItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e) {
      int result;
      result = fileSelection.showOpenDialog(
                              JMenuBarDemo.this);

      if (result == JFileChooser.APPROVE_OPTION) {
         File f = fileSelection.getSelectedFile();
         JOptionPane.showMessageDialog(JMenuBarDemo.this,
              "Open file selected was " + f.getPath());
      }
   }});
```

Open file selection dialog

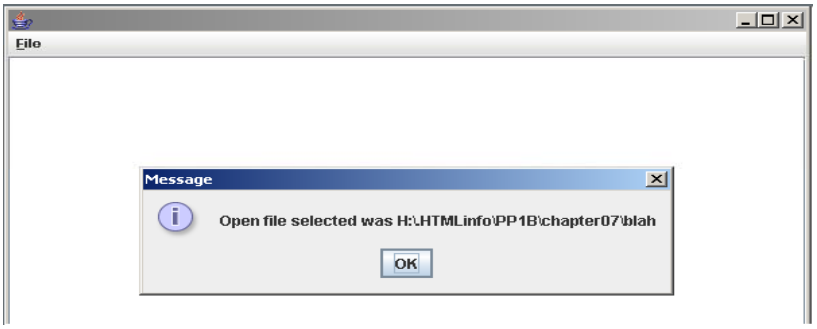
Process file if user clicks on "Open" button

JMenuBarDemo - Using a file selection dialog (cont)

The File Selection Dialog



The Dialog box



JMenuBarDemo example (cont)

```
saveItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e) {
      int result;

      result = fileSelection.showSaveDialog(
                              JMenuBarDemo.this);

      if (result == JFileChooser.APPROVE_OPTION) {
         File f = fileSelection.getSelectedFile();
         JOptionPane.showMessageDialog(JMenuBarDemo.this,
              "Save file selected was " + f.getPath());
      }
   }});

 closeItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
       fileOpen = false; text.setText("");
       contentPane.remove(scroll);
       contentPane.repaint();
}});
```

Registering listeners to the save/close buttons

JMenuBarDemo example (cont)

```
      exitItem.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e) {
            System.exit(0);
         }});


      fileMenu.add(newItem);  fileMenu.add(openItem);
      fileMenu.add(saveItem); fileMenu.add(closeItem);
      fileMenu.add(exitItem); fileMenu.addSeparator();

      menu.add(fileMenu);  this.setJMenuBar(menu);

      contentPane = this.getContentPane();
      contentPane.setBackground(Color.gray);

      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
   }// end constructor
```

JMenuBarDemo example (cont)

```
   public static void main (String [] args) {
      JMenuBarDemo demo = new JMenuBarDemo();

      demo.setSize(600,400);
      demo.setLocation(250,250);
      demo.setVisible(true);
   }
} // end class
```

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 7: User Interfaces and Graphics III
Software Design Issues, Model View Controller (MVC)

Lecture Slides

Implementing a GUI based system

- One of the primary goals of *implementing* a good interactive (GUI based) system is to separate the user interface code from the application code.
- Benefits:
  - Interface can be more easily modified (portability, accessibility etc.)
  - Application code can change without affecting the interface (e.g. text file based data is replaced by DBMS)
  - Project can be more easily split across teams (requires good management, communication)
  - Provides a logical basis for modularising the application

Model View Controller

- One technique for modularising an interactive system is the Model View Controller approach:
  - Originally conceived for the Smalltalk environment
  - Currently used in other popular systems such as the Microsoft Foundation Classes (MFC) for Visual C++
  - Used internally within AWT/Swing
  - Can be implemented easily and effectively in Java with communication between Packages, Classes and Listeners
  - Increases *cohesion* while managing *coupling* in a structured way

Model View Controller

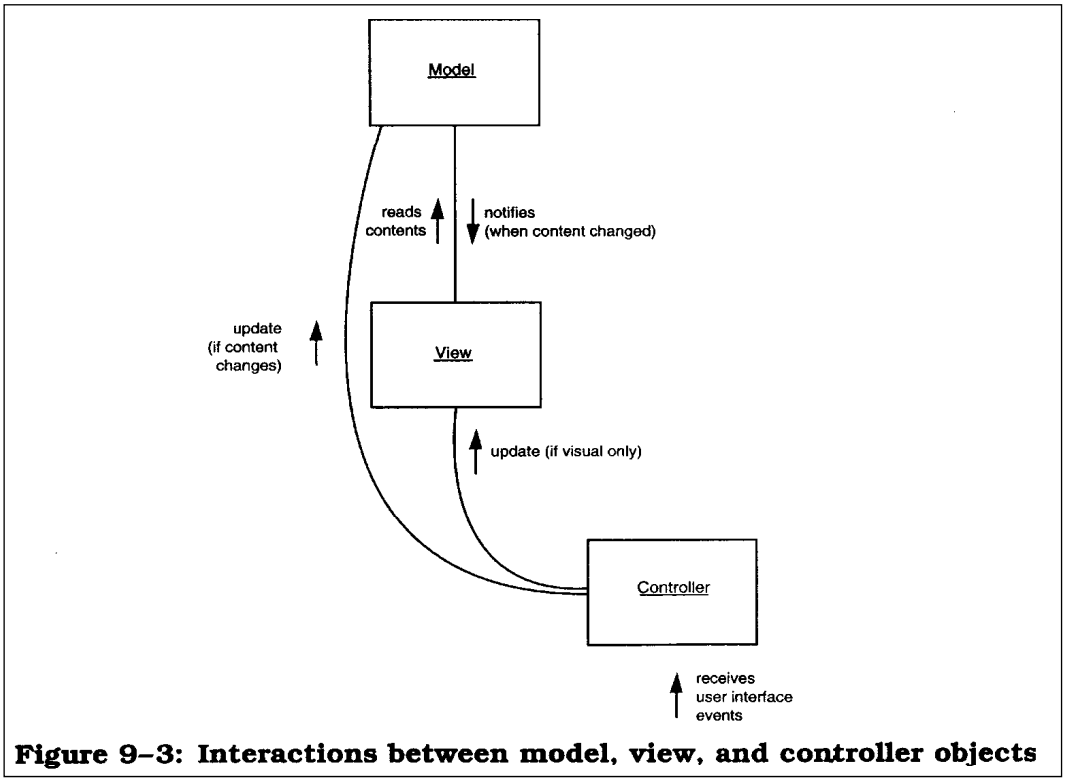


Figure 9–3: Interactions between model, view, and controller objects

Model View Controller

- Model: contains application code such as:
  - **data structures** **I/O routines**
  - **accessor methods** **calculations etc.**
- View: the representation of the data i.e. how it is presented to the user
  - (may be devices other than simple display terminal e.g. accessibility devices for disabled persons)
  - multiple views of the same data (e.g. a spreadsheet has cell and chart views)
- Controller: handles user interaction and mediates between the Model (data) and The View (representation)
  - e.g user drags mouse in a drawing program. The controller modifies the series of points or vectors (data structures) in the model as the mouse is moved (user interaction) and instructs the view to redraw the data in the model (alternatively the model may automatically instruct it's registered views to redraw its data - example of Observer pattern [Design Patterns, Gamma et al. 1995])

Model View Controller

- The Model, View and Controller are treated as separate modules
  - may be classes or packages depending upon their complexity
  - a system may have multiple views, controllers and models
  - multiple views are quite common but not required
  - nearly always a separate controller for each view (for cohesion) although these are often implemented as inner classes in Java and thus not entirely separate
  - multiple models are usually used when there exist groups of independent program data

Inner Classes

- Version 1.1 of the JDK introduced the notion of inner classes
  - often used with event handling
  - simplify the relationship between event sources and listeners
- Non-static (standard) inner classes:
  - have direct access to the implementation of their enclosing class (including private attributes and methods)
  - are automatically initialised with a reference to the instance of the outer class that created it [usually the reference is invisible and implicit e.g. `SomeMethodCall()` but can be made explicit with `EnclosingClassName.this.SomeMethodCall()`]

Inner Classes

- Inner classes may be named or anonymous
- Static inner classes are similar to C++ nested classes
- Can be hidden from other classes in a package (private)
- The next five slides show five variations on the relationship between event sources (`ThreeDButton` in this case) and listeners and shows how inner classes can be helpful in simplifying this relationship

Separate Listener

```
class ThreeDButtonListener extends    MouseAdapter
                           implements MouseMotionListener {
   public void mousePressed(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();
        button.paintBorderInset();
   }
   public void mouseClicked(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();
        button.paintBorderRaised();
   }
   public void mouseReleased(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();
        button.paintBorderRaised();
   }
   public void mouseDragged(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();

        if(button.contains(event.getX(), event.getY())) {
               if(button.getState() == ThreeDButton.BORDER_RAISED)
                      button.paintBorderInset();
        }
        else {
               if(button.getState() == ThreeDButton.BORDER_INSET)
                      button.paintBorderRaised();
        }
   }
   public void mouseMoved(MouseEvent event) {}
}
```


Separate Listener (Event Source)

```
class ThreeDButton {
   public ThreeDButton
   {
        ThreeDButtonListener ThreeDlistener=new ThreeDButtonListener();
        addMouseListener(ThreeDlistener);
        addMouseMotionListener(ThreeDlistener);
   }

   public void paintBorderInset()
   {
        // ...
   }
   public void paintBorderRaised()
   {
        // ...
   }
}
```

Separate Listener

```
class ThreeDButtonListener extends    MouseAdapter
                           implements MouseMotionListener {
   public void mousePressed(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();
        button.paintBorderInset();
   }
   public void mouseClicked(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();
        button.paintBorderRaised();
   }
   public void mouseReleased(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();
        button.paintBorderRaised();
   }
   public void mouseDragged(MouseEvent event) {
        ThreeDButton button = (ThreeDButton)event.getSource();

        if(button.contains(event.getX(), event.getY())) {
                if(button.getState() == ThreeDButton.BORDER_RAISED)
                        button.paintBorderInset();
        }
        else {
                if(button.getState() == ThreeDButton.BORDER_INSET)
                        button.paintBorderRaised();
        }
   }
   public void mouseMoved(MouseEvent event) {}
}
```


Separate Listener

- Advantages:
  - self contained and cohesive
  - can have multiple sources use a single instance of a listener if desirable
  - listener is not ‘hard coded’ to the source

  => source can be modified independently of listener
- Disadvantages:
  - must call `getSource()` in every method that needs access to the event source
  - may be cumbersome to create separate classes for very small one line operations

Separate Listener Alternative (Modified Event Source)

```
class ThreeDButton {
   public ThreeDButton
   {
        ThreeDButtonListener ThreeDlistener=new ThreeDButtonListener(this);
        addMouseListener(ThreeDlistener);
        addMouseMotionListener(ThreeDlistener);
   }

   public void paintBorderInset()
   {
        // ...
   }
   public void paintBorderRaised()
   {
        // ...
   }
}
```


Separate Listener (alternative)

```
class ThreeDButtonListener extends    MouseAdapter
                          implements MouseMotionListener {
   private ThreeDButton button;
   public ThreeDButtonListener(ThreeDButton button) {
        this.button=button;
   }
   public void mousePressed(MouseEvent event) {
        button.paintBorderInset();
   }
   public void mouseClicked(MouseEvent event) {
        button.paintBorderRaised();
   }
   public void mouseReleased(MouseEvent event) {
        button.paintBorderRaised();
   }
   public void mouseDragged(MouseEvent event) {
        if(button.contains(event.getX(), event.getY())) {
                if(button.getState() == ThreeDButton.BORDER_RAISED)
                        button.paintBorderInset();
        }
        else {
                if(button.getState() == ThreeDButton.BORDER_INSET)
                        button.paintBorderRaised();
        }
   }
   public void mouseMoved(MouseEvent event) {   }
}
```

Separate Listener (alternative)

- Advantages:
  - maintains reference to source [does not need to call `getSource()`]
  - source can be modified independently of listener

- Disadvantages:
  - may be inefficient if we must construct separate instances for multiple sources

Combined Source/Listener

Example 9-21 ThreeDButton Listening to Itself

```
class ThreeDButton extends    Canvas
                   implements MouseListener, MouseMotionListener {
    static public  int BORDER_INSET = 0, BORDER_RAISED = 1;
    int state = BORDER_RAISED;
    ...
    public void mousePressed (MouseEvent event) {
        paintBorderInset();
    }
    public void mouseClicked (MouseEvent event) {
        paintBorderRaised();
    }
    public void mouseReleased(MouseEvent event) {
        paintBorderRaised();
    }
    public void mouseDragged(MouseEvent event) {
        if(contains(event.getX(), event.getY())) {
            if(state == ThreeDButton.BORDFR_RAISED)
                paintBorderInset();
        }
        else {
            if(state == ThreeDButton.BORDER_INSET)
                paintBorderRaised();
        }
    }
    public void mouseEntered(MouseEvent event) { }
    public void mouseExited (MouseEvent event) { }
    public void mouseMoved  (MouseEvent event) { }
}
```

Combined Source/Listener

- Advantages:
  - no separate class is necessary
  - less lines of code

  → Really advantageous?
- Disadvantages:
  - reduced cohesion
  - reduced extensibility (cannot modify source independently of listener)
  - classes may become large and more difficult to maintain

Think carefully before using this approach!

Named Inner Class

Example 9-22 ThreeDButton with Inner Classes for Event Handling

```
class ThreeDButton extends Canvas {
    static public  int BORDER_INSET = 0, BORDER_RAISED = 1;
    int state = BORDER_RAISED;

    public ThreeDButton() {
        addMouseListener       (new ThreeDButtonMouseListener());
        addMouseMotionListener(
            new ThreeDButtonMouseMotionListener());
    }
    class ThreeDButtonMouseListener extends MouseAdapter {
        public void mousePressed (MouseEvent event) {
            paintBorderInset();
        }
        public void mouseClicked (MouseEvent event) {
            paintBorderRaised();
        }
        public void mouseReleased(MouseEvent event) {
            paintBorderRaised();
        }
    }
    class ThreeDButtonMouseMotionListener
                extends MouseMotionAdapter {
        public void mouseDragged(MouseEvent event) {
            if(contains(event.getX(), event.getY())) {
                if(state == ThreeDButton.BORDER_RAISED)
                    paintBorderInset();
            }
            else {
                if(state == ThreeDButton.BORDER_INSET)
                    paintBorderRaised();
            }
        }
    }
```

Named Inner Class

- Advantages:
  - can automatically reference event source (because it is the enclosing class)
  - does not need to call `getSource()` or maintain a reference (as in the two separate examples)
  - maintains fairly good encapsulation and cohesion

- Disadvantages:
  - more difficult to share coding responsibilities (since they are in the same code module)
  - assumes a one to one mapping between source and listener instances

Anonymous Inner Class

Example 9-23 ThreeDButton with Anonymous Inner Classes for Event Handling

```
class ThreeDButton extends Canvas {
    static public  int BORDER_INSET = 0, BORDER_RAISED = 1;
    int state = BORDER_RAISED;

    public ThreeDButton() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed (MouseEvent event) {
                paintBorderInset();
            }
            public void mouseClicked (MouseEvent event) {
                paintBorderRaised();
            }
            public void mouseReleased(MouseEvent event) {
                paintBorderRaised();
            }
        });
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent event) {
                if(contains(event.getX(), event.getY())) {
                    if(state == ThreeDButton.BORDER_RAISED)
                        paintBorderInset();
                }
                else {
                    if(state == ThreeDButton.BORDER_INSET)
                        paintBorderRaised();
                }
            }
        });
```

Anonymous Inner Class

- Advantages:
  - provides a syntactic shortcut by combing the instantiation with the definition
  - very convenient for small handler methods
  - can automatically reference event source (because it is the enclosing class)
  - does not need to call `getSource()` or maintain a reference (as in the two separate examples)
- Disadvantages:
  - code clarity and cohesion may be reduced (some people may disagree!)
  - more difficult to share coding responsibilities (since they are in the same code module)
  - assumes a one to one mapping between source and listener instances
  - anonymous classes can only be instantiated once

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 8: Linked Lists, Basic Searching and Sorting

Lecture Slides

Dynamic Data Structures

So far, all data collections used have been either arrays or Java's built in collection classes.

To understand how such data structures and collection classes are implemented this lecture discusses the **linked list** (the simplest of self-referential data structures, with sequential access.)

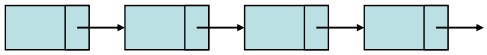
Self-Referential Structures

Many dynamic data structures are implemented through the use of a **self-referential structure**.

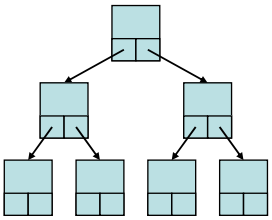
A self-referential structure is an object, one of whose elements is a reference to **another object of its own type**.

With this arrangement, it is possible to create 'chains' of data of varying forms:

DataNode
String data;
int moreData;
DataNode next;



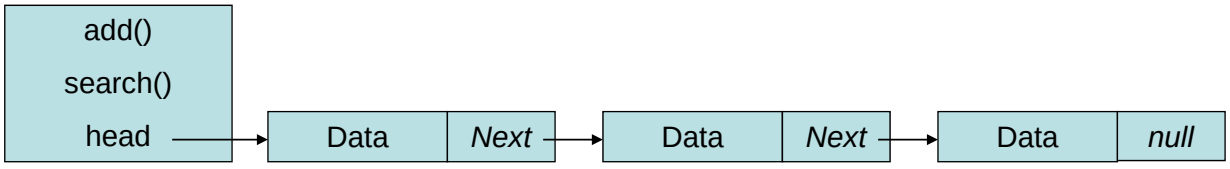
lists



trees

Linked Lists: Introduction

The simplest self-referential dynamic data structure is the **linked list**. It is simply a chain of self-referential data nodes.



The node comprises **two** items; one (or more) attributes of data, and the pointer to the next data node, commonly referred to as the **`next`** reference.

The **end of the list** is defined by a **`null`** next reference. The start (or **`head`**) of the list is contained in a **`header` class**, which also encapsulates all the list's functionality. This helps the linked list become an independent, cohesive package. The header class provides the API to access the list.

Linked Lists: the `List` Node

The list node is a simple self-referential structure that stores an item of data, and a reference to the next item.

```
class ListNode {

  private int data;

  ListNode next;


  public ListNode(int data){
    this.data = data;
    this.next = null;
  }
}
```

The data variable is where the information to be stored resides. It may be of any primitive or reference type but is usually generic (java.lang.Object or parameterised type <T>).

The next variable is the self-referential link to the next data item.

The constructor initialises the node object by storing the data that was given as an argument, and sets the next reference to **`null`**.

Linked Lists: the `header` Class

The **`header`** class is the public interface for the linked list. It is where the functionality is stored (as methods), and contains a link to the first item of the list (the 'head'.)

```
class List {

  private ListNode head;


  List() {
    head = null;
  }


  add();    // discussed later
  find();   // discussed later
  delete(); // discussed later
}
```

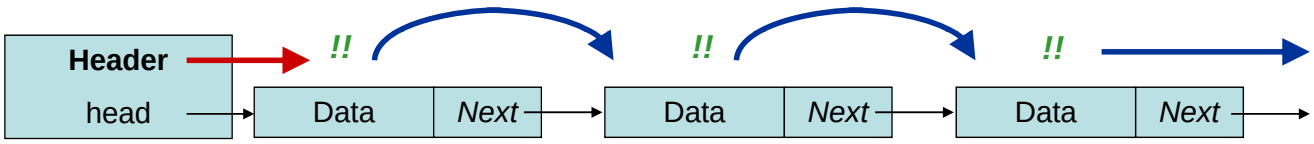
The `head` variable is a reference to the first item in the list.

The constructor initialises the list by setting the head to `null` (an empty list.)

The methods provide a way to use the list. They each access the structure through the head reference.

Linked Lists: List Traversal (1)

It is sometimes necessary to traverse the entire length of the list to perform some function (for example, to count the number of items, or display summary information.)



Step 1: Step through the list from the header node forward.

Step 2: Perform the desired operation at that node.

Step 3: Move onto the next node, until the end of the list is reached.

List traversal forms the basis of many of the list manipulation operations such as add, retrieve and delete.

Linked Lists: List Traversal (2)

The code below will traverse the entire list, and print out the data contained in each node.

```
public void traverse() {
    ListNode current = head;

    while (current != null) {

        System.out.println(current.data);

        current = current.next;
    }
}
```

Step 1: Maintain a variable to store the current position in the list.

Step 2: Continue stepping through the list, until the end of the list (a **`null`** reference) is reached.

Step 3: At each step, print out the data present in the current node.

Because of the way a single link list is defined, we can only access data in one direction, and sequentially (one item after another.)

Linked Lists: Adding Data

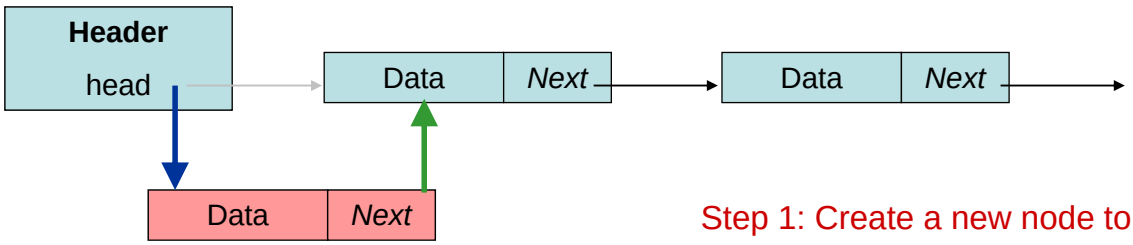
Data is added to a linked list by wrapping the data to add into a node, and then placing that node at the appropriate place in the data structure. Note that the API of the list would usually hide the ‘wrapping’.

Depending on the circumstances and purpose of the list, there are a number of places where data may be added:

- **At the start (head) of a list**
- **In the middle of the list (e.g. at a specific index or node count)**
- **At the end (tail) of the list**
- **At the appropriate place to preserve sort order**

Linked Lists: Adding Data to the Head (1)

Adding data to the head of a list is the easiest and quickest way in which it can be done.



Step 1: Create a new node to store the given data.

Step 2: Set the new node’s next reference to the first node.

Step 3: Reset the head reference to point to the newly created node.

The order in which the link manipulations are done are very important; they must always be done from right to left, otherwise data nodes will be lost.

Could use temporary variables and do in any order but is less elegant.

Linked Lists: Adding Data to the Head (2)

Here is the code for the previous slide:

```
public void addToHead(ListNode aNode) {

  aNode.next = head;

  head = aNode;
}
```

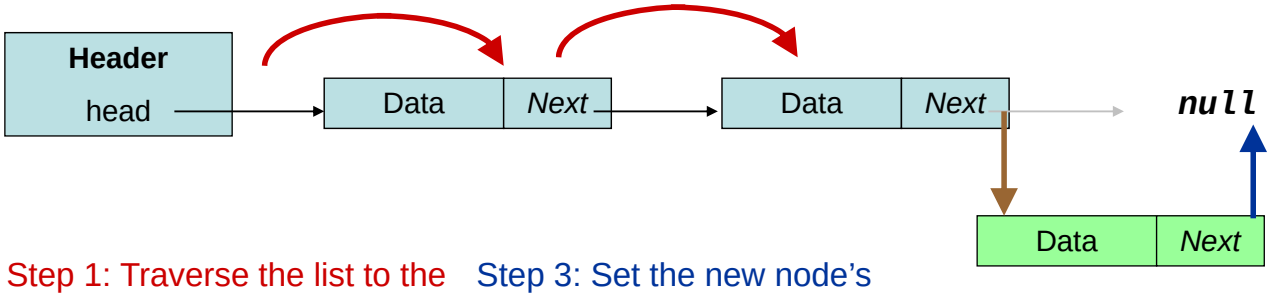
And you would use it thus:

```
ListNode newNode = new ListNode(data);  // call constructor

addToHead(newNode);                     // call method
```

Question: Does the caller really need to know about the ListNode class?

Linked Lists: Adding Data to the Middle or Tail (1)

Adding data to the middle (e.g. at a specific index or node count) or tail of the list is essentially the same process. The diagram below illustrates adding to the end (tail.)



Step 1: Traverse the list to the desired insertion point (in this case, the last item of the list.)

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

Step 4: Reset the insertion point's next reference to point to the newly created node.

Linked Lists: Adding Data to the Middle or Tail (2)

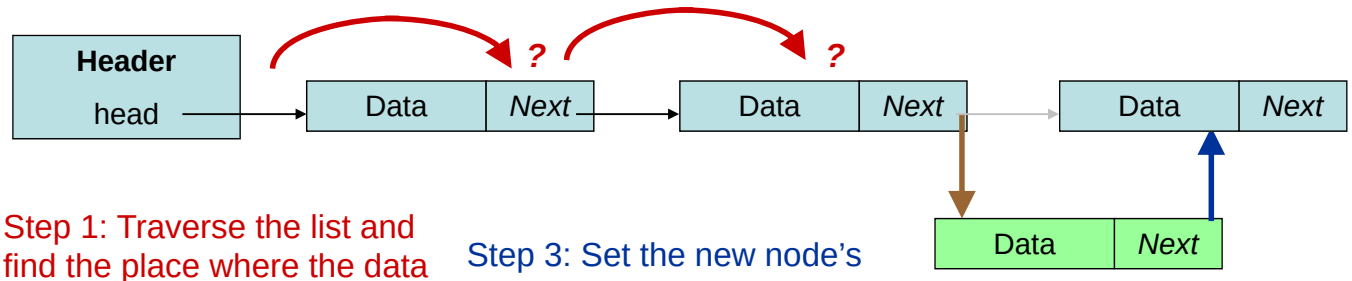
```
public void addToTail(ListNode aNode) {

  ListNode insert = head;
  // check for empty list
  if (head == null)
    head = aNode;
  else
  {
    // could instead check for index here if
    // inserting in middle of list
    // while we are not at the end of the list
    while (insert.next != null)
       insert = insert.next;

    aNode.next = null;
    insert.next = aNode;
  }
}
```

And you use it:

```
ListNode newNode = new ListNode(data);

addToTail(newNode);
```


Linked Lists: Adding Data in Sort Order (1)

Adding data in sort order is somewhat more complex. This is because we do not know beforehand which insertion strategy (head, middle, tail) is needed, and must be determined on-the-fly.



Step 1: Traverse the list and find the place where the data should be stored. Both 'current' and 'previous' references will need to be maintained.

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

Step 4: Reset the insertion point's next reference to point to the newly created node.

Note: Steps 3 and 4 will need to be implemented differently if the insertion point is the head.

Linked Lists: Adding Data in Sort Order (2)

```
public void addInOrder(ListNode aNode) {
  ListNode current = head;
  ListNode previous = null;
   while ((current != null) && (current.data <= aNode.data)) {
    previous = current;
    current = current.next;
  }
  // head
  if (current == head) {
    aNode.next = head;
    head = newNode;
  }
  // tail
  else if (current == null) {
    previous.next = aNode;
  }
  // middle
  else {
    previous.next = aNode;
    aNode.next = current;
  }
}
```

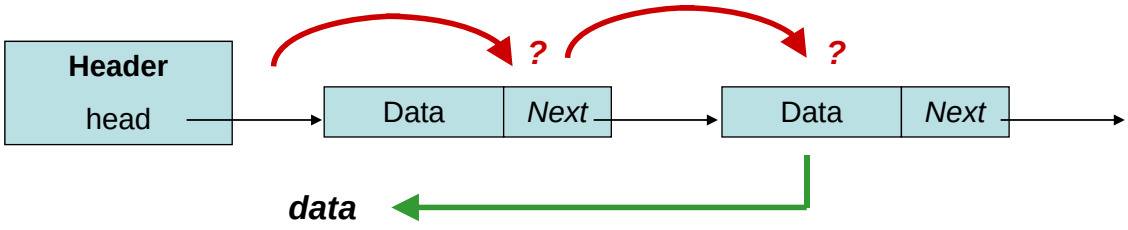

Linked Lists: Adding Data in Sort Order (3)

And you use it:

```
ListNode newNode = new ListNode(data);

addInOrder(newNode);
```

Linked Lists: Retrieving Data (1)

Data retrieval consists of traversing the data structure until a requested node is found (e.g. a specific value or an item at an index). If the data is not found, some form of failure signal should be returned instead (e.g. boolean return value).



Step 1: Traverse the list.

Step 2: While traversing, compare the search 'key' value with the data in each node.

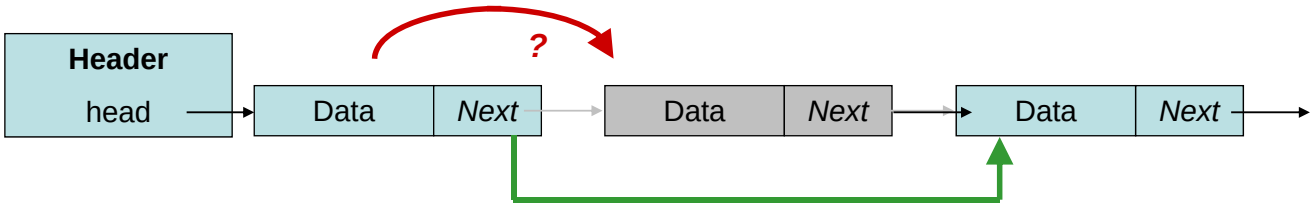
If the data keys match, return the data portion of the node.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Linked Lists: Deleting Data (1)

Deletion is very similar to retrieval. As before, the list is traversed to find data matching a given 'key' value. However, instead of returning the data, the node is to be deleted.

The node can be deleted by having the next references 'jump over' the node to delete. To do this, the node before the one to delete must be known, and as such, the traversal needs to keep track of two references.



Step 1: Traverse the list, maintaining both current and previous references.

Step 2: If the search key matches the current data node, 'jump over' the current node, and return success.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Data Storage, Retrieval and Sorting

One of the primary applications of computers is the **storage** (through data structures) **and retrieval** (via **search** algorithms for those structures) **of data**.

Unorganised data is easier to manage (since data additions and deletions can be done easily) but slower to **search** (since entire collection must be searched using **linear search**)

Therefore where data retrieval speed is important, data should be sorted or indexed (e.g. hashing) in some way.

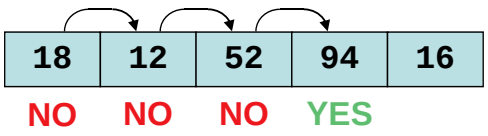
Common sorting algorithms include quick sort, merge sort, insertion sort, selection sort and bubble sort. Large data sets are often sorted on disk rather than in memory using techniques such as b-tree (not covered in this course).

Selection sort is covered in this lecture to provide a simple example.

The Linear Search

The **linear search** algorithm is the simplest of all searches. It merely **iterates through the collection**, checking for a match between the data at that position and the key. Only useful for unsorted/non-indexed data (use Binary Search otherwise).

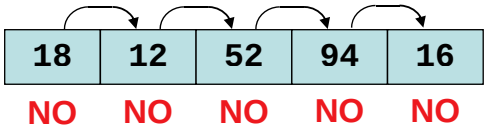
*(looking for **94**)*



Result: *index 3*

If the data is **not present** in the collection, the linear search will look through **all elements**, and **finish with no result.**

*(looking for **78**)*



Result: *failure*

The Linear Search Algorithm

The **linear search algorithm** psuedocode is simply:

```
loop i from 0 to the last element
   if collection[i] is equal to the target
      return i
   end if
end loop
return -1
```

The basic algorithm is similar for a **linked list**. However, instead of looping through element indices, the loop will continue moving through the list nodes **until the end of the list has been reached.**

The Big-O Notation

A common way of assessing algorithm complexity is **Big-O** notation. This is an evaluation of how running time increases as the volume of data (N) increases.

A crude Linear Search may examine *every* item in the collection. Therefore, running time has a direct linear relationship to volume of data. We call this a performance of **O(N)**, where N is the volume of data. This means that the complexity of the search is linearly related to the size of the data. Examples of Big-O(N): N, 3N, 1/2 N, N + 3, but not N^2, N^3, Log(N), 10^N

Big-O means that the complexity *is an approximation of the specified N*

A Linear Search should stop when the correct key is found.

- ***Best*** case is when the key is in the first position. A performance where only one operation is needed is **O(1)**.
- ***Worst*** case is when the key is in the last position and all items must be examined: **O(N)**.
- ***Average*** case is when the key is about halfway, giving a performance of **O(N/2)**.

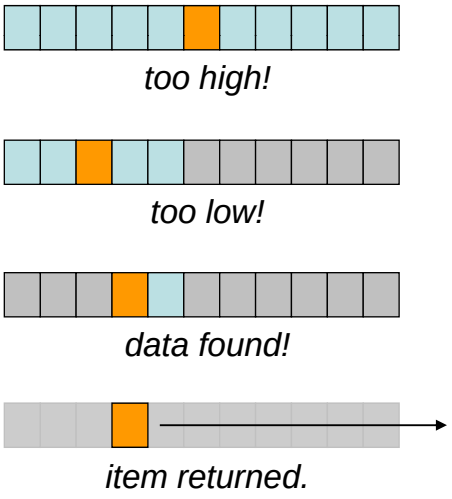
Java API Arrays.sort() algorithm uses Merge Sort which has **O(N * log(N))** performance

The Binary Search

The **binary search** algorithm is far more efficient **O(log N)** than the linear search. However, **it relies on two conditions**: the data that is being searched **sorted and is indexed**.

It works by looking at the **middle element of the collection**; depending on whether that item is **larger** or **smaller than the data searched for**, it **disregards the other half** of the collection.

This is **repeated** until the target data has been **tracked down**, or the **boundaries are invalid** (the data does **not exist**.)



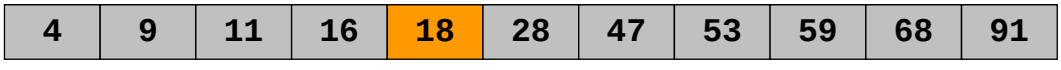
The Binary Search: An Example

Consider the collection below, searching for **18**:

***28** is too **high** – the **right hand** half is discarded.*

***11** is too **low** – the **left hand** half is discarded.*

***16** is too **low** – the **left hand** half is discarded.*



*The target (**18**) has been **found**.*

The Binary Search: Another Example

Consider the collection below, searching for **37**:

***28** is too **low** – the **left hand** half is discarded.*

***59** is too **high** – the **right hand** half is discarded.*

***47** is too **high** – the **right hand** half is discarded.*

*There is **no more data** – therefore, the target is **not present**.*

The Binary Search Source Code

```
public static int binarySearch(int[] numbers, int target)
{
   int index, left = 0, right = numbers.length - 1;

   while (left <= right)
   {
      index = (left + right) / 2;

      if (target == numbers[index])
         return index;

      if (target > numbers[index])
         left = index + 1;
      else
         right = index - 1;
   }

   return -1;
}
```

Ensure the boundaries are valid.

Find the mid-point.

If a match is found, return the position.

Otherwise, reset the boundaries to ignore the irrelevant half.

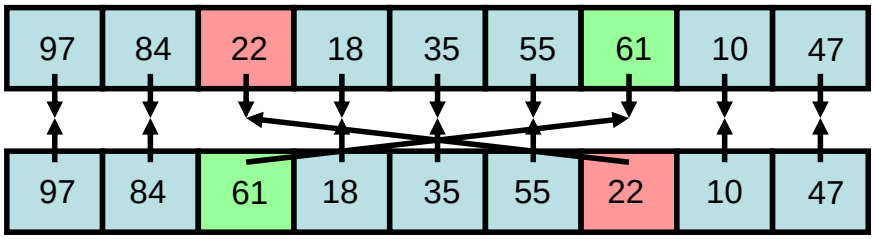
Sorting Algorithms

The performance of a sorting algorithm is affected by:

1. machine time needed for running the coded algorithm
2. memory requirements (due to recursion, see topic 9)

(2) is not always as critical since most searching algorithms have a memory requirement directly related to the size of data. (If records are large, they can be sorted *indirectly* by sorting the keys and references.)

When a key is moved, so is the reference to the rest of the record, which stays as is.



** (1) and (2) are interrelated. In general, the more complex sorting algorithms need less machine time. If sorting is needed infrequently, or there is only a small amount of data to be sorted, it is better to use a simple method. For small data sets, they are sometimes more efficient.

Selection Sort (1)

The **selection sort** algorithm gets its name, because it works on the **successive selection of items** in a particular order.

Consider a selection sort algorithm to sort a collection of integers in smallest to largest order.

The first step is to find the **smallest piece of data** in the collection, and **swap it into the first** position in the collection .

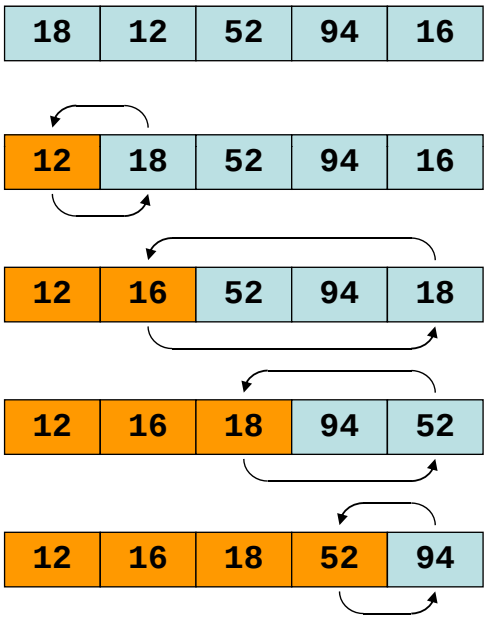
It will then find the smallest data present **in the remaining section** of the collection, and swap it with the second position.

This process continues, finding the smallest data in the (shrinking) subsection of unprocessed data, until there is **only one data item left.**

Best case is when the data are already sorted; we can avoid unnecessary exchanges (as with bubble, so performance is **O(N)**. The worst case, data are in inverse order, still has a performance no greater than **$O(N^2)$**.

The Selection Sort (2)

Consider the collection below:



This is the collection, before the selection sort commences.

It finds **12** as the smallest number, and swaps with **index 0**.

It finds **16** as the smallest number, and swaps with **index 1**.

It finds **18** as the smallest number, and swaps with **index 2**.

It finds **52** as the smallest number, and swaps with **index 3. The collection is now sorted.**

Selection Sort: Source Code

```
public static void selectionSort(int[] numbers)
{
   int min, temp;

   for (int i = 0; i < numbers.length-1; i++)
   {
      min = i;
      for (int scan = i+1; scan < numbers.length; scan++)
         if (numbers[scan] < numbers[min])
            min = scan;

      // swap the values
      temp = numbers[min];
      numbers[min] = numbers[i];
      numbers[i] = temp;
   }
}
```

RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 9: Recursion, Binary Search Trees

Lecture Slides

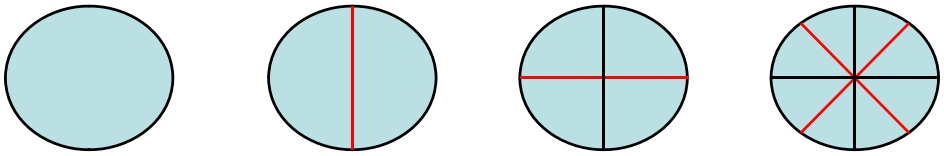
Recursion

Recursion is the process of defining something in terms of itself. The aim is to continually define a problem in terms of a simpler (or partial) form of itself, until the result is simple enough to be directly evaluated (e.g. without iteration).
The result then passes back through the previous definitions, with the answer being built up at each stage. By the time the result falls back to the original problem, the proper solution has been evaluated.

Recursion: A Simple Conceptual Example

Consider the process of cutting a cake into equal sized slices.

First, the cake is cut in two. Then, each half is cut into quarters. Each quarter is cut into two eighths .. (etc.)
This process continues until the desired slice size is obtained.



This cake cutting was a recursive process: the only task that is being done is cutting in half. The cake was first cut in half, then the halves were cut in half, and so on.

Parts of a Recursive Algorithm

Recursive algorithms are defined by two kinds of conditions:

- a recursive step
- a terminating condition

In the cake example, the *recursive step* was the act of cutting each cake section in half.
The *terminating condition* was to stop cutting when the desired cake slice size was reached.

It is possible that there may be many recursive steps and terminating conditions for a given algorithm.

A Mathematical Example (1)

A factorial of a number *x* is the product of all values from 1 up to and including *x*.
For example:

$4! = 4 \times 3 \times 2 \times 1 = 24$

(an exclamation mark is the symbol for factorial.)

The conventional iterative form of the operation is:

$x! = x \times x\text{-}1 \times x\text{-}2 \ldots \times 1$

This can also be expressed recursively as:

$x! = x \times (x\text{-}1)!$

A Mathematical Example (2)

We could use this recursive definition to calculate the result of 4 factorial:

$4! = 4 \times 3!$

And we could then use the same definition to complete the calculation:

$3! = 3 \times 2!$

$2! = 2 \times 1!$

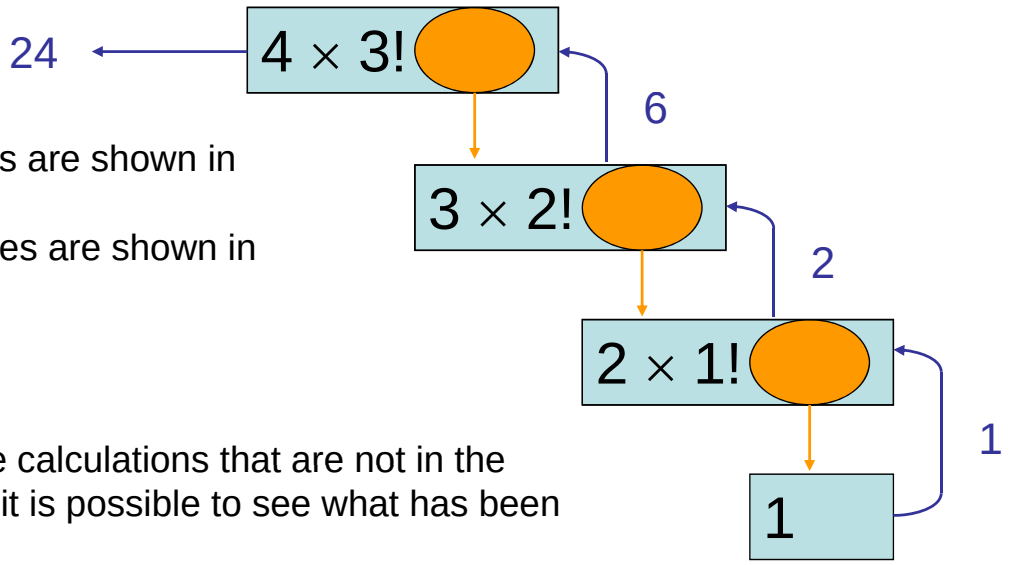
Recursive Step:
$x! = x \times (x\text{-}1)!$
Terminating Condition:
$x = 1$

1 factorial is simply 1 – it is the terminating condition (there are no more recursive steps required to arrive at a complete solution):

$1! = 1$

A Mathematical Example (3)

It is easier to see how the recursion works to arrive at the result diagramatically:



Recursive calls are shown in orange.
Returned values are shown in blue.

By reading the calculations that are not in the orange circle, it is possible to see what has been calculated.

A Mathematical Example in Java Code

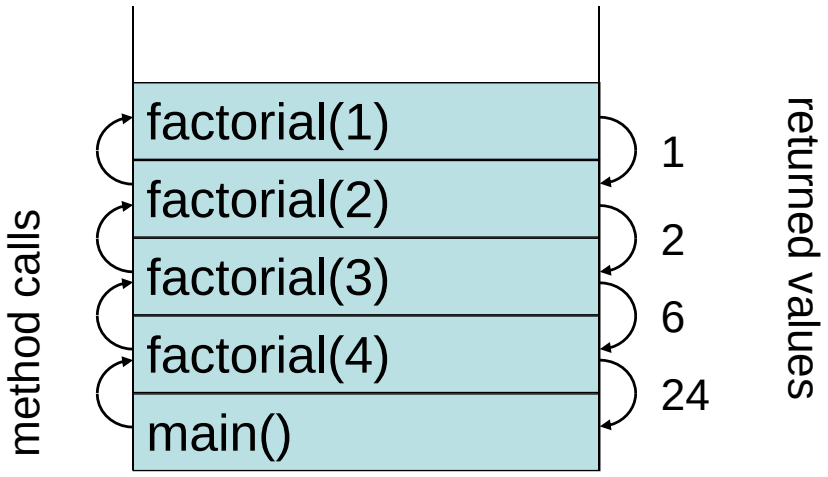
Once the recursive and terminating steps have been defined, an implementation is fairly easy to write:

```
public int factorial(int a) {
  if (a == 1)
    return 1;
  else
    return (a * factorial(a-1));
}
```

The terminating condition is highlighted in red, and the recursive step in blue.

Recursion and the Method Stack

Each recursive call to a method is placed on top of the method stack.
Calculating 4 factorial with the given source code can be visualised as:



Recursion vs. Iteration

Both recursion and iteration have their benefits and disadvantages.

Recursion is commonly found in a number of mathematical situations. It can also be a useful technique for the design of particular algorithms to manage strings, lists and other types of data structures.
Iteration is more efficient and preferable when there is a simple iterative (loop based) solution for a problem (for example, the simple factorial operation used as an example for recursion could be implemented more efficiently using iteration.)

Recursion should also be avoided for long calculations where the method stack may overflow.

Recursion vs. Iteration: Efficiency

A Fibonacci number is the sum of the previous two Fibonacci numbers. For example, 1,1,2,3,5,8,13,21,34,55,89, …

```
public static long fibonacci(int n)
{
  if (n <= 1)
    return 1;
  else
    return (fibonacci (n – 1) + fibonacci (n – 2);
}
```

```
int fib1 = 1, fib2 = 1, temp=0;
for (int i = 1; i < n; i++)
{
  temp=fib2;
  fib2=fib1+fib2;
  fib1=temp;
}
return fib2;
```

In this case, the iterative solution is more efficient than the recursive one.
Some algorithms lend themselves to iteration, others to recursion.

Thinking Recursively: String Length

The length of a string is 1 plus the length of the rest of the string.
The length of a string is 0 when the string is empty.

Recursive Step: if the string isn't empty, length = 1 + length of the remainder of the string.
Terminating condition: if the string is empty, the length is 0.

```
static int stringLen(String s)
{
  if (s.equals(""))
    return 0;
  else
    return (1 + (stringLen(s.substring(1))));
}
```

Thinking Recursively: String Reversing (1)

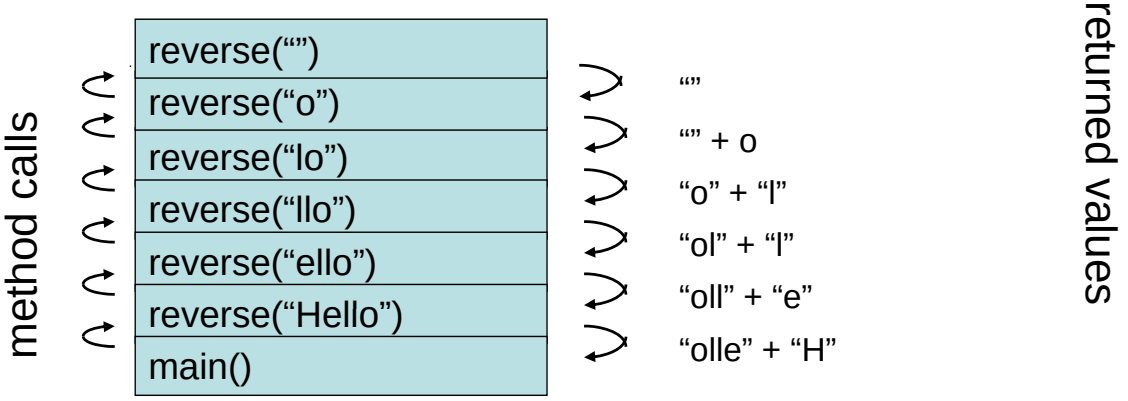
A reversed string is the reverse of all characters except the 1st, with the first character appended to the end.
A reversed string is empty when the string is empty.

Terminating condition: if the length is zero, return a blank string.
Recursive Step: return the reverse of the String (from the second character onwards), with the first character appended to the end of the result.

```
static String stringRev(String s)
{
    if (s.length()==0)
        return ("");
    else
        return (stringRev(s.substring(1)) + s.charAt(0));
}
```

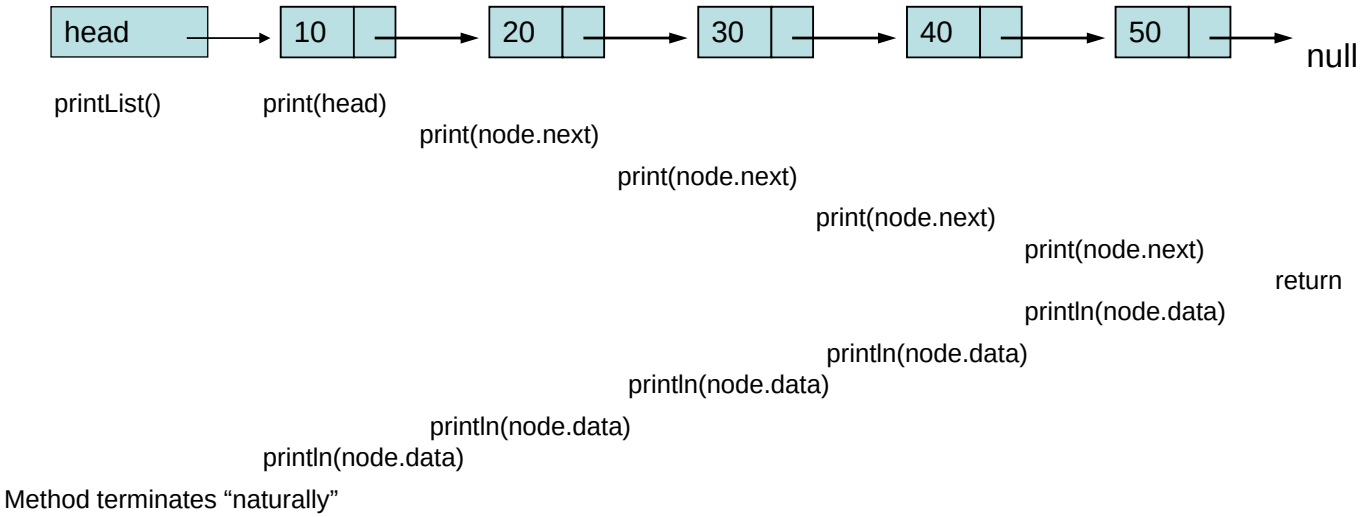

Thinking Recursively: String Reversing (2)

Again each recursive call to a method is placed on top of the method stack.
Calculating 4 factorial with the given source code can be visualised as:



Thinking Recursively: Linked List (1)

Let's look at another example – printing a linked list in reverse order.
The terminating condition will be when we reach the NULL value at the end of the list and the recursive call will effectively move to the next "node" in the list.
Once we have reached the end of the list we can print the values stored in the nodes we have visited.



Thinking Recursively: Linked List (2)

We still want to keep the details of now the data is stored in the ADT hidden, so what we have to do is add a method to the public interface for printing the list in reverse and then have this public interface method call our private recursive print method (which effectively remains hidden from the client program).

```
private void printReverse(ListNode head)
 {
   if (head != null)
   {
     System.out.println(head.number);
     printReverse(head.next);
   }
 }

private void print(ListNode head)
 {
   if (head != null)
   {
     print(head.next);
     System.out.println(head.number);
   }
 }
```

Stop calling method recursively when end of list is reached.

Print data in current node before subsequent recursive calls have been completed

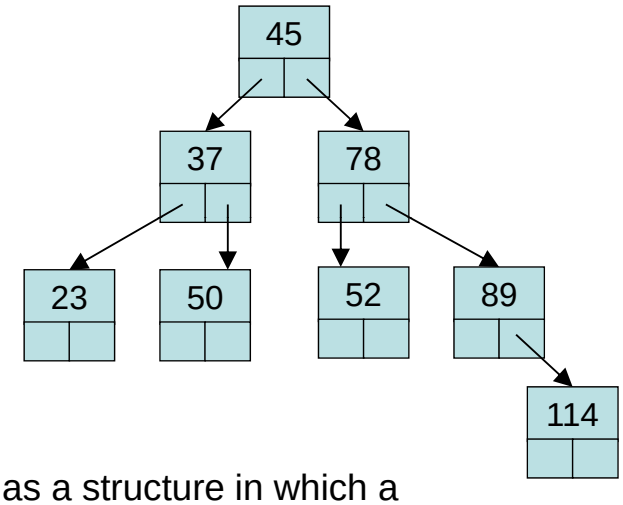
Call method again to move to next node in list

Print data in current node after subsequent recursive calls have been completed

Tree Structures

Linked lists, vectors, stacks and queues are linear (sequential) data structures. A tree is different; it is non-linear, or hierarchical.

The tree structure is naturally suited to many applications (e.g. databases, file systems, web sites, etc.), and can often allow algorithms to be significantly more efficient.



A tree (in data structure terms) is defined as a structure in which a set of *nodes* are connected together by their *edges*, in a parent-child relationship.

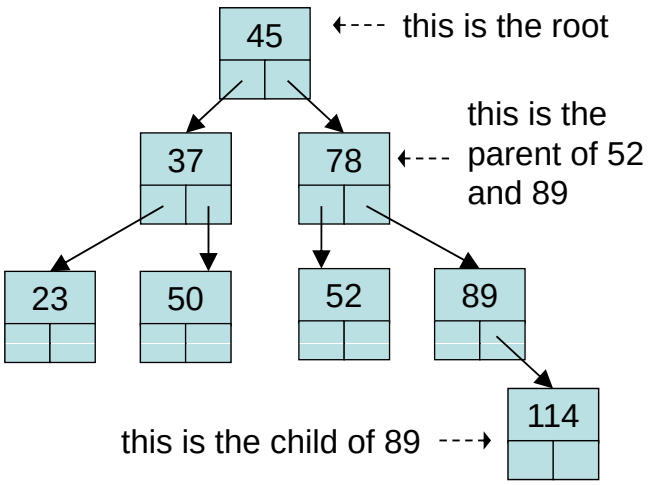
(Visualising the 'tree' relationship may be easier looking at the diagram upside-down, with the root at the bottom.)

Tree Terminology (1)

A **tree** has a single distinguished node called the *root*, and every other node is connected to only one parent (c.f. a graph).

A **binary tree** is one in which each node has at most two children, each called the *left* and *right* child.

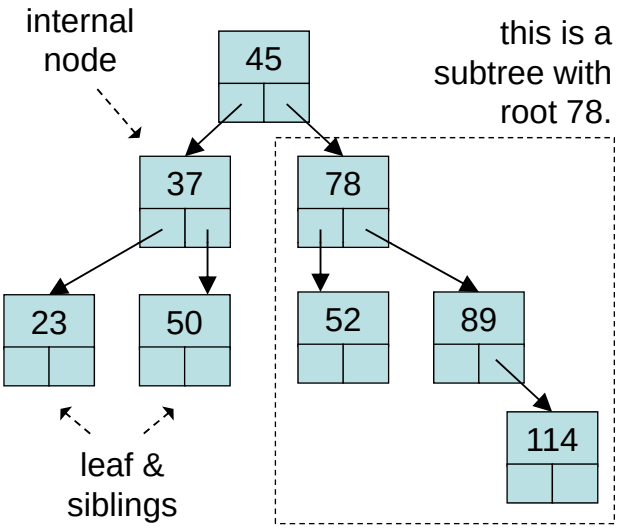
The diagram to the right is an example of a binary tree.



A recursive definition of a tree is as follows; a tree is either empty, or consists of a root node, with zero or more *subtrees* as children (read on.)

Tree Terminology (2)

- nodes with the **same parent** are called **siblings (e.g. 37 & 78)**
- an **leaf (external, or terminal) node** has **no children (e.g. 23 & 50)**
- an **internal (non-terminal) node** has **one or more children (e.g. 37 & 78)**
- if there is a path from node A to node B, then A is an **ancestor** of B, and B is a **descendant** of A.

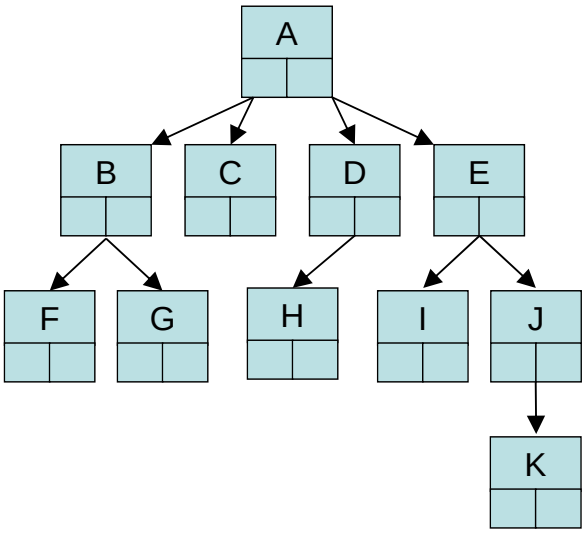


A **subtree** of a tree, rooted at a particular node, is a tree consisting of that node and all its descendants.

This reveals that a tree is inherently a **recursive structure**.

Tree Properties

- the **depth** of a node is the length of the path from the root to the node. (The root has a depth of 0.)
- the **height** of a node is the length of the path from the node to the deepest leaf.
- the **size** of a node is the number of nodes in the subtree rooted at that node (including itself.)



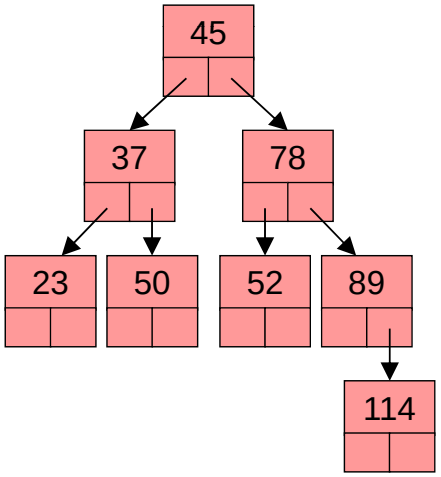
	A	B	C	D	E	F	G	H	I	J	K
Depth	0	1	1	1	1	2	2	2	2	2	3
Height	3	1	0	1	2	0	0	0	0	1	0
Size	11	3	1	2	4	1	1	1	1	2	1

Binary Search Tree (1)

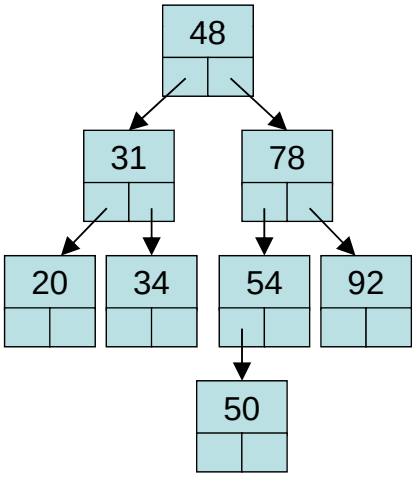
A **binary tree** is any tree with at most two descendents per node

A **binary search tree** is a tree (not automatically balanced) in which:

- all descendants **to the left** of any node have **lesser** data values
- all descendants **to the right** have **greater** data values.



not a binary search tree



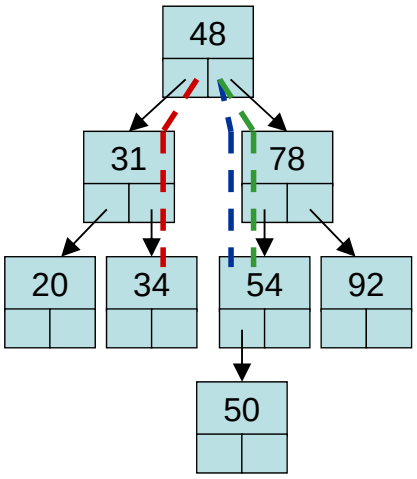
a binary search tree

Binary Search Tree (2)

A binary search tree satisfies the search order property, and can be seen as **an extension of the binary search algorithm** to allow insertions and deletions.

Best case is ***O(log N)*** as in binary search (when the tree is balanced)

Worst case is ***O(N)*** as in linear search (when the tree is linear, like a linked list)



to search for 34, we need to go only through nodes 48, 31 and 34.

to insert 60, we need to go only through nodes 48, 78 and 54.

to remove 54, we need to through nodes 48, 78 and 54. But, we also need to **reattach the tree** to maintain the ordering!

Binary Search Tree ADT (1)

Let us now implement an Abstract Data Type (ADT) for a binary search tree, and go through the algorithms for inserting, finding, traversing and removing elements.

First we must identify the basic operations to create the interface:

- **find** – find a node in the tree
- **insert** – insert a node into the tree
- **remove** – remove a node from a tree
- **printPreorder** – traverse and print every node in *pre-order*
- **printInorder** – traverse and print every node in *in-order*
- **printPostorder** – traverse and print every node in *post-order*

Binary Search Tree ADT (2)

Nodes in a binary search tree ADT must encapsulate data values of a generic type. But, **which generic class to use?**

Every class is a descendant of **`java.lang.Object`**, however unlike stacks or queues, where we were simply storing and removing elements, **we need to do comparisons** on the data contained inside the nodes.

We need a generic class that can be **compared in three ways** (equals, less than, and greater than) – one such interface is **`java.lang.Comparable`**. This method includes a method called **`compareTo()`**, which can be used for such comparisons.

Therefore, any object stored in the BST **will need to implement the `Comparable` interface.** Another restriction is that the BST's methods must use only parameters of, and return, **`Comparable`** data types.

The user of the BST **must use an object that implements `Comparable`** that is appropriate for their application.

NOTE: See refactored topic9 source code for example of using parameterised generic type and improved encapsulation

Binary Search Tree ADT Specification (1)

First, we define the interface generic to all types of binary tree.

```
interface BinaryTree
{
   public Comparable find (Comparable target);
   public boolean insert (Comparable item);
   public boolean remove (Comparable item);
   public void printPreorder ();
   public void printInorder ();
   public void printPostorder();

}
```

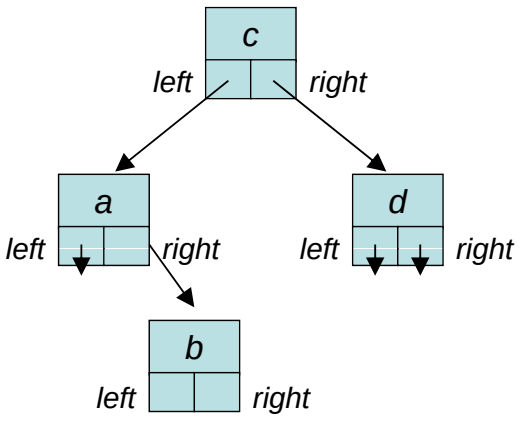
The implementation can be private since the user does not need to see it in order to use the specified methods.

Binary Search Tree ADT Node (1)

Recall the implementation of a linked list…

A tree has nodes, similarly to a linked list.

However, unlike a linked list, a tree has more than one 'next' reference; in the case of a binary tree, two references: left and right child node.



Also, recall that we need to use the interface *Comparable* for the type of the data value in the node (*see next slide.*)

Binary Search Tree ADT Node (2)

```
class BinaryNode
{
   Comparable data; // see Caspar's refactored BinaryNode
   BinaryNode left; // for better encapsulation/generics
   BinaryNode right;

   public BinaryNode(Comparable item)
   {
      data = item;
      left = null;
      right = null;
   }
}
```

Initialise the two child references to point nowhere (no children.)

Class BinaryNode and its attributes are package visible (default modifiers), so that the binary tree implementation (presumably in the same package) can use it. It is not made public, since it should only be accessible by other tree classes. (c.f. Caspar's refactored topic 9 code which makes the node class private)

Binary Search Tree ADT Public Implementation (1)

```
public class BinarySearchTree implements SearchTree
{
   protected BinaryNode root;

   public Comparable find(Comparable target) throws
                                                ItemNotFound
   {
      return find(target, root);
   }

   public void insert (Comparable item) throws DuplicateItem
   {
      root = insert (item, root);
   }

   public void remove (Comparable item) throws ItemNotFound
   {
      root = remove(item, root);
   }
```

These are calls to protected internal methods that we will define shortly.

Binary Search Tree ADT Public Implementation (1)

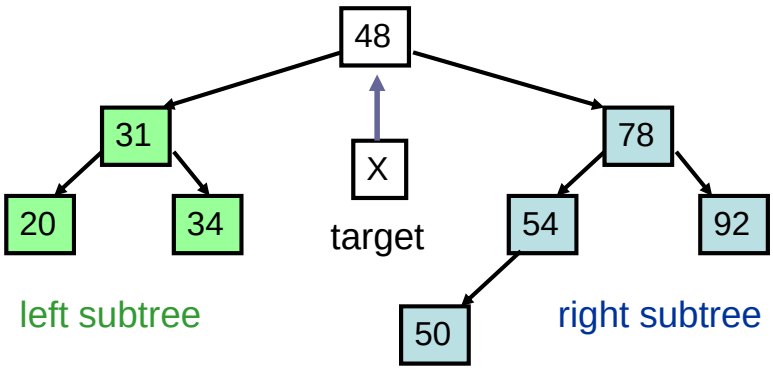
```
    public void printPreorder()
    {
       preorder(root);
    }

    public void printInorder()
    {
       inorder(root);
    }

    public void printPostorder()
    {
       postorder(root);
    }
}
```

These are calls to protected internal methods that we will define shortly.

All these methods rely on internal methods to actually do the work. We will now investigate what these internal methods have to do with the tree structure, and implement them in Java code.

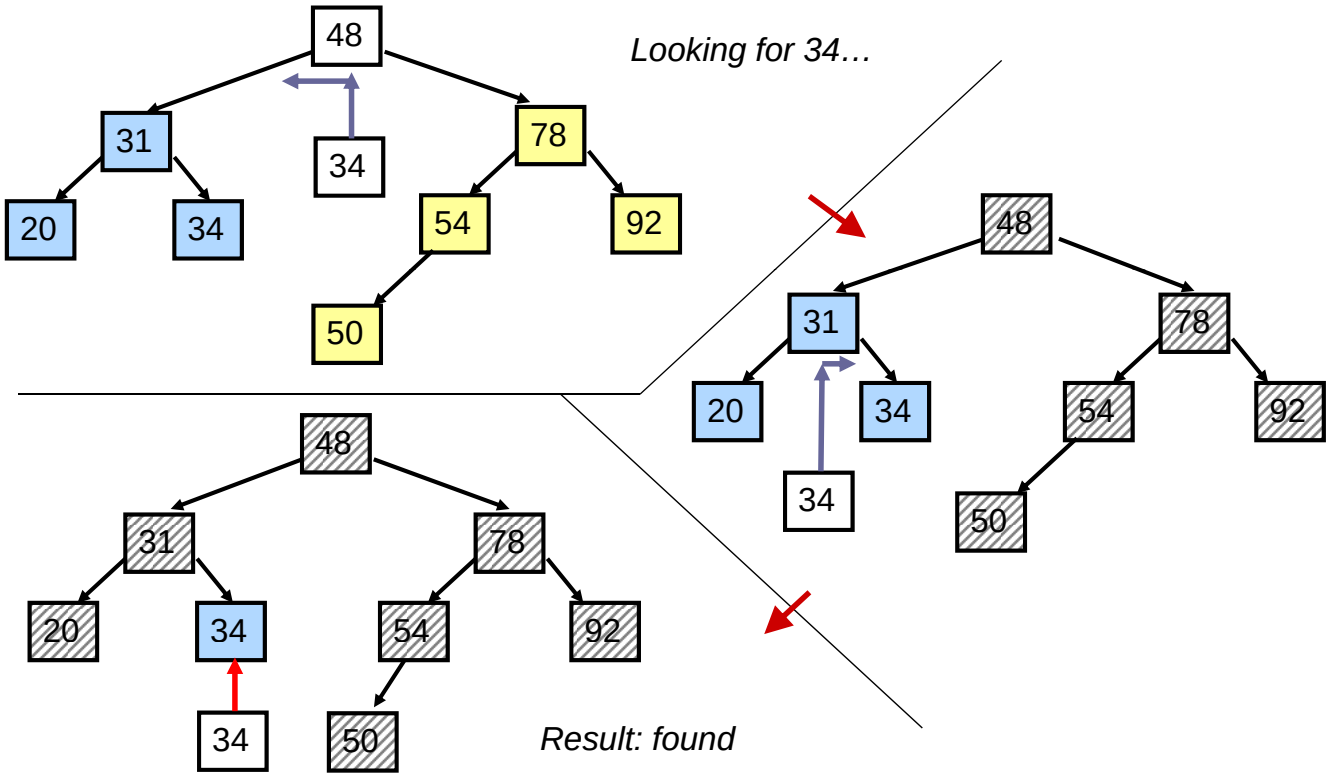
Binary Search Tree: Find

Recall the binary search algorithm, and apply it here:



To search for **X** in a tree with root **R**,
compare **X** with the data value $\mathbf{R_v}$ of **R**
if **X** = $\mathbf{R_v}$ success - terminate
else if X < $\mathbf{R_v}$ make the root of the left subtree of **R** the new **R**
else if X > $\mathbf{R_v}$ make the root of the right subtree of **R** the new **R**
Repeat the above until success or finish.
The binary search finishes without success when **R** is null (cannot go left or right)

Binary Search Tree: Find (an example)



Binary Search Tree ADT: Find Code

```
protected Comparable find(Comparable x, BinaryNode r)
                                        throws ItemNotFound
{
   while (r != null)
   {
      if (x.compareTo (r.data) < 0)
         r = r.left;
      else if (x.compareTo (r.data) > 0)
         r = r.right;
      else
         return r.data;
   }

   throw new ItemNotFound("Find fails");
}
```

Iterate through the branches.

Move through the tree as necessary.

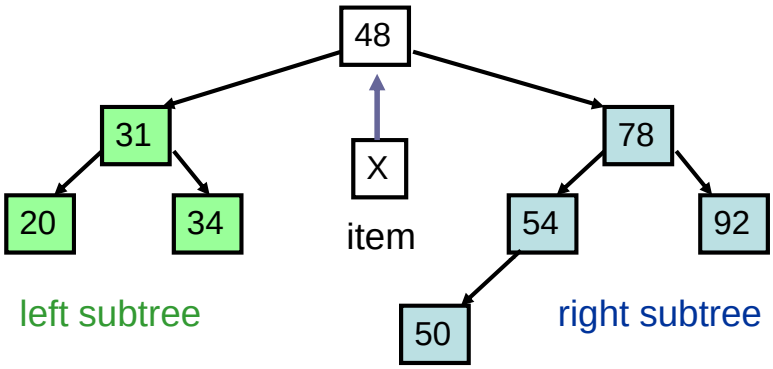
Throw an exception if the find operation failed.

The throwing of exceptions at a find failure is purely a design decision – you could, for example, make it return a null reference instead*. However what if data is meant to be null? Returning node instead of data avoids this problem.

Binary Search Tree: Insert

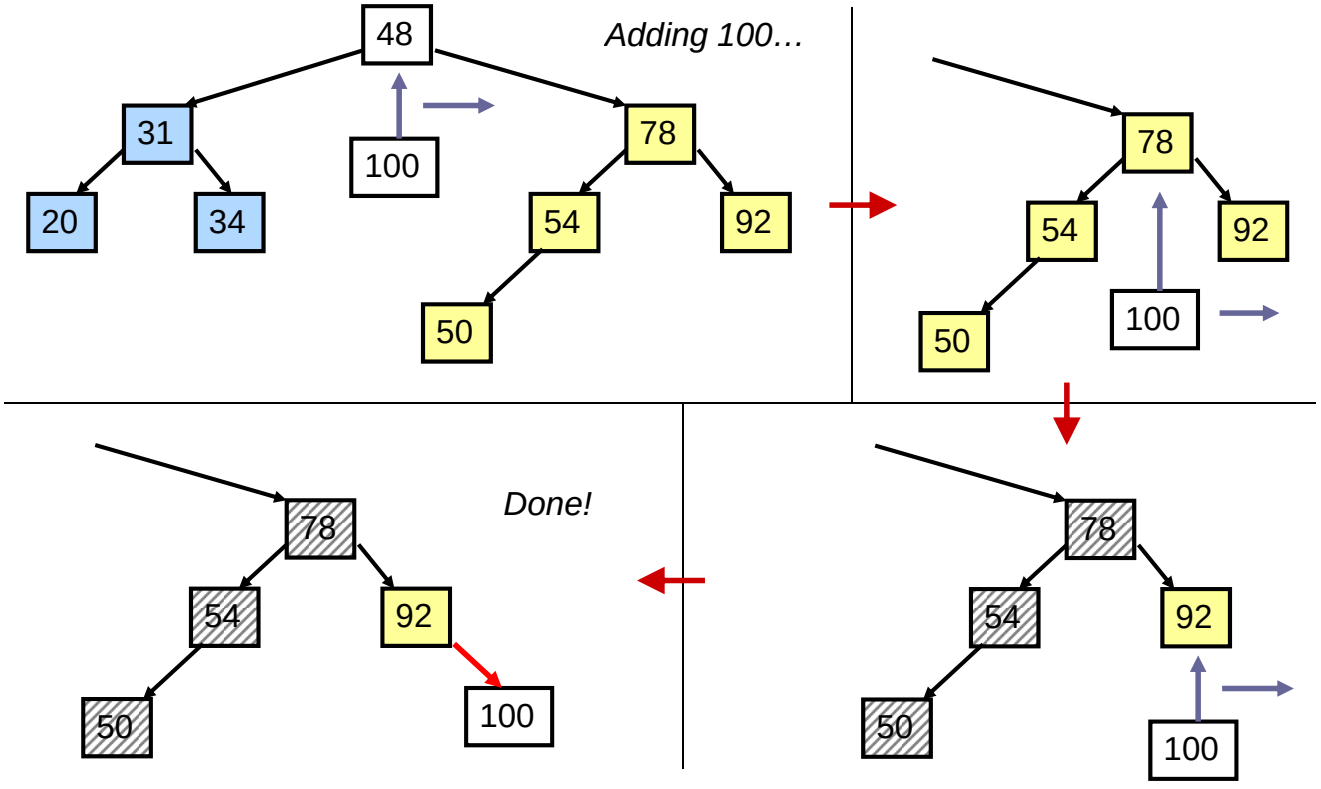
Insertion can be done recursively:

(Note how simple the code ends up being…)



To insert a value **X** into a tree with root **R** that stores a data value $\mathbf{R_v}$
if the tree is empty simply set its root **R** to the new node – recursion returns
else if $\mathbf{X} < \mathbf{R_v}$, insert the value in the left subtree
else if $\mathbf{X} > \mathbf{R_v}$, insert the value in the right subtree
else if $\mathbf{X} = \mathbf{R_v}$, throw a duplicate item exception – recursion returns
Recursion returns when either the new node has been inserted or a duplicate is found

Binary Search Tree: Insert (an example)



Binary Search Tree ADT: Insert Code

```
protected BinaryNode insert(Comparable x, BinaryNode r)
                                        throws DuplicateItem
{
   if (r == null)
      r = new BinaryNode (x);
   else if (x.compareTo (r.data) < 0)
      r.left = insert (x, r.left);
   else if (x.compareTo (r.data) > 0)
      r.right = insert (x, r.right);
   else
      throw new DuplicateItem("Duplicate item attempt");

   return r;
}
```

We create a new node, to save the existing root node given (remember, r is just a local parameter variable)

Move through the tree if necessary.

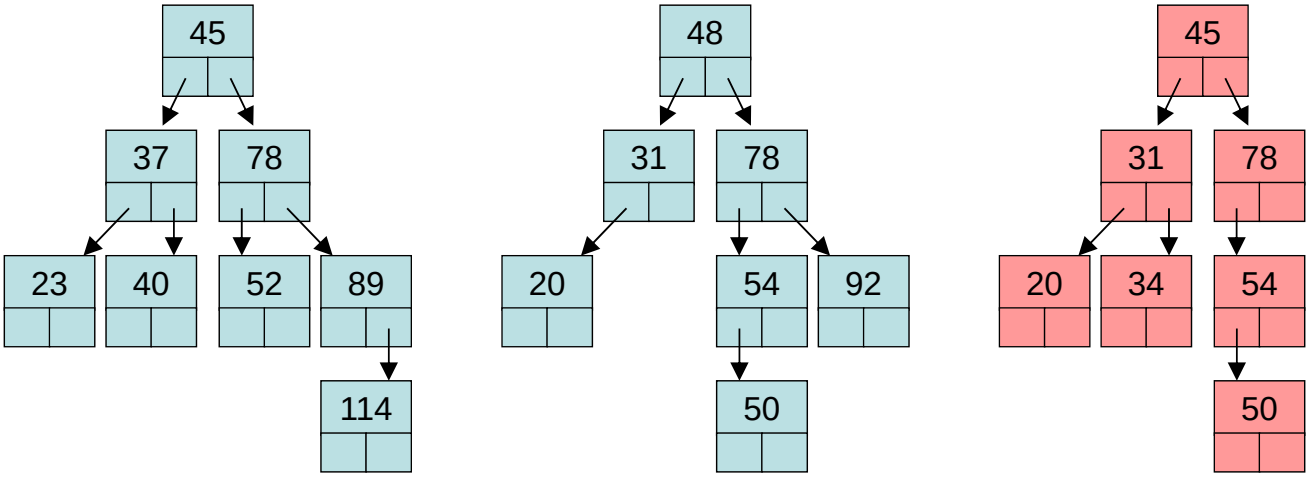
We need to return the new node reference.

In the publically-available insert method, the root is changed by assigning the protected insert's return value to it.

Balanced Binary Tree

A binary tree is balanced if and only if, for every node, the height of its left and right subtree differ by at most 1.

Some trees are self-balancing, e.g. AVL trees (inventors Adelson, Velskii, and Landis) and red-black trees (used in java.util.TreeMap/TreeSet).



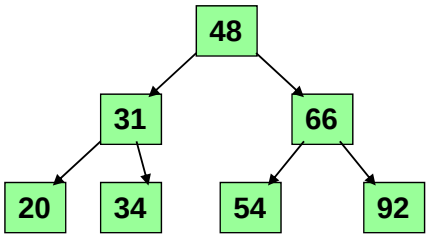
balanced tree *balanced tree* *unbalanced tree*

Binary Search Tree: Find and Insert Performance

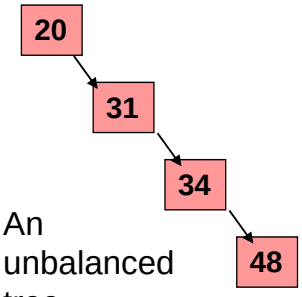
• If the tree is perfectly balanced, the situation is exactly like in binary search. A tree of **N** = 2^x nodes takes maximum **x** comparisons. That is, we have logarithmic access cost, **O(logN)**

• If the tree is unbalanced, the situation tends towards linear search. The worst case is when the tree degenerates to a mere linked list. In this case a tree of **N** nodes takes maximum **N** comparisons. That is, we have linear access cost, **O(N)**

• Binary search tree is worst to store sorted data values and is best to store random data values.



A balanced tree



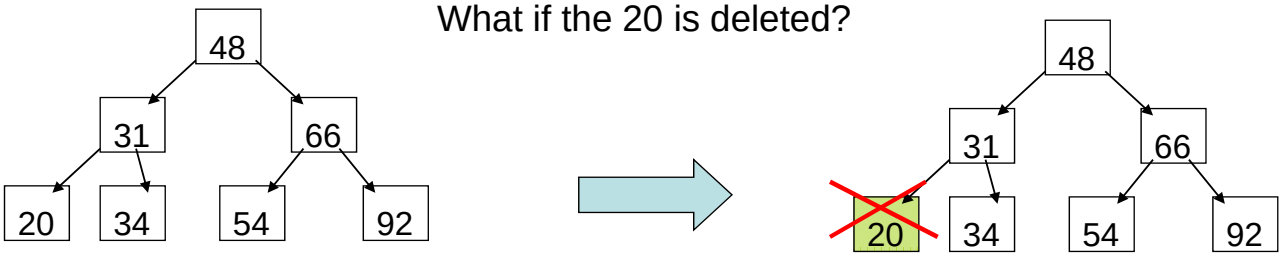
An unbalanced tree

Binary Search Tree: Deletion (1)

Removing an element from a BST can be broken down into 3 cases.

Case 1: If the node to be removed is a leaf (*i.e. no children*), it can be deleted immediately.

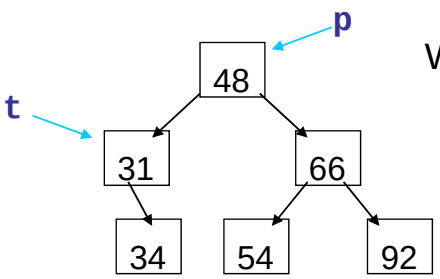
What if the 20 is deleted?



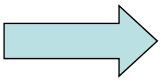
Binary Search Tree: Deletion (2)

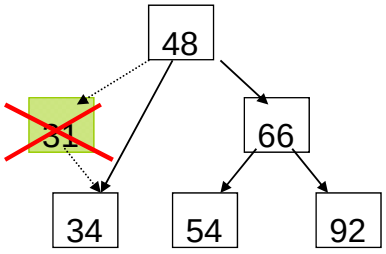
Case 2: If the node to be removed has one child…

the node can be deleted by altering the link from its parent to itself, to point to its child.



What if the 31 is deleted?





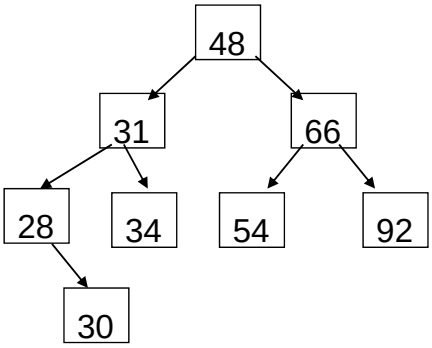

```
p.left = t.right
```


Binary Search Tree: Deletion (3)

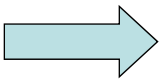
Case 3: If the node to be removed has two children…

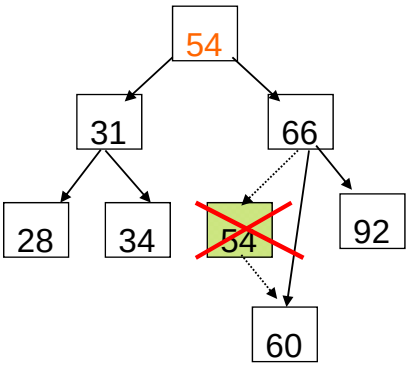
One way to remove it is by replacing the data of this node with the smallest data of the **right** sub-tree.

Then the node with the smallest data is removed (since the smallest data node can only have at most one node it can be handled as case 1 or case 2).



What if the 48 is deleted?





Binary Search Tree: Deletion Code (1)

```
protected BinaryNode remove (Comparable x, BinaryNode r)
                                          throws ItemNotFound
{
   if (r == null)
      throw new ItemNotFound("target not found to delete");

   if (x.compareTo(r.data) < 0)
      r.left = remove(x, r.left);
   else if (x.compareTo(r.data) > 0)
      r.right = remove(x, r.right);

   else if(r.left != null && r.right != null)
   {
      r.data = (findMin( r.right )).data;
      r.right = remove( r.data, r.right);
   }
```

Find the target node to delete, moving through the tree as necessary

Deletion process if there are two children

Binary Search Tree: Deletion Code (2)

```
   else
   {
      if (r.left != null)
         r = r.left;
      else
         r = r.right;
   }
   return r;
}

BinaryNode findMin (BinaryNode r)
{
   if (r == null)
      return null;
   else if (r.left == null)
      return r;

   return findMin (r.left);
}
```

Deletion process if there is at most one child node.

Find the smallest data in a tree by moving as far left as possible.

Traversing a Binary Tree

Traversing a tree means systematically processing every node in that tree.

For a binary tree, there are two links for each node; therefore, we have three basic orders in which we can process the tree:

- *preorder:* process the node, then visit the left and right subtrees
- *inorder:* visit the left subtree, then process the node, and then the right subtree
- *postorder:* visit the left and right subtrees, and then process the node

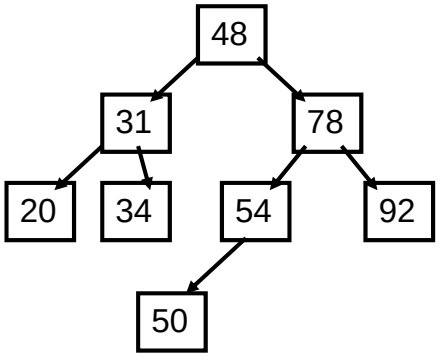
Every node is accessed, therefore performance is **O(*N*)**

Traversal: Pre-Order

```
protected void preorder (BinaryNode r)
{
   if (r == null) return;
   System.out.print (r.data + " ");
   preorder (r.left);
   preorder (r.right);
}
```

Output: 48 31 20 34 78 54 50 92

Note: The root is at the start.

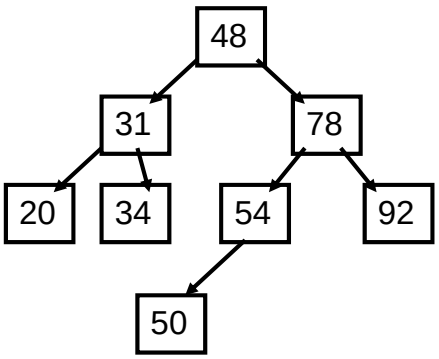


Traversal: In-Order

```
protected void inorder(BinaryNode r)
{
   if (r == null)
      return;

   inorder (r.left);
   System.out.print (r.data + " ");
   inorder (r.right);
}
```

Output: 20 31 34 48 50 54 78 92

Note: output is sorted for binary search tree



Traversal: Post-Order

```
protected void postorder(BinaryNode r)
{
   if (r == null)
      return;

   postorder (r.left);
   postorder (r.right);
   System.out.print (r.data + " ");
}
```

Output: 20 34 31 50 54 92 78 48

Note: The root is at the end